(1)

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

(1)

AFIT/GCS/ENG/88D-3

A Vectorized Hidden-Surface Algorithm Implemented on the Cray-2 Supercomputer

THESIS

Roy Donehower
Captain, USAF

AFIT/GCS/ENG/88D-3

AFIT/GCS/ENG/88D-3

A VECTORIZED HIDDEN-SURFACE ALGORITHM IMPLEMENTED ON THE CRAY-2 SUPERCOMPUTER

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Engineering

Roy Donehower, B.S.

Captain, USAF

December, 1988

Preface

I'd like to thank Major Phil Amburn for his superb job as a thesis advisor. I am also indebted to Mr. Bob Conley at the Air Force Weapons Laboratory, who was instrumental in my completing this thesis. He listened to my thesis ideas patiently for countless hours and contributed many of his own ideas. Finally, I would like to express my gratitude to Dr. Larry Rapagnani of the Air Force Weapons Laboratory for sponsoring this thesis, and providing the Cray-2 computer time as well as the travel funds to implement my thesis at the Weapons Laboratory and to attend SIGGRAPH '88.

Roy Donehower

Table of Contents

Page

Preface ii

Table of Contents iii

List of Figures iv

List of Tables v

Abstract vi

I. **Introduction** 1-1

1.1 **Introduction** 1-1

1.2 **Background** 1-1

1.2.1 The Popularity of Graphics Workstations 1-1

1.2.2 The Need to Visualize Data Produced by Supercomputers 1-2

1.2.3 The Future of Computer Graphics 1-3

1.2.4 Supercomputer Graphics Research 1-4

1.3 **Thesis Statement** 1-5

1.4 **Assumptions** 1-5

1.5 **Scope** 1-6

1.6 **Approach/Methodology** 1-6

1.7 **Materials/Equipment** 1-7

1.8 **Conclusion** 1-7

Page

II. **Literature Search** 2-1

2.1 **Introduction** 2-1

2.2 **The Cray-2 Supercomputer** 2-1

2.2.1 Architecture 2-1

2.2.2 Cray-2 Memory Organization 2-3

2.2.3 The Advantages of Vectorization 2-5

2.2.4 The Advantages of Pipelining 2-5

2.2.5 Vectorizing Compilers 2-8

2.2.6 Graphics Supercomputers 2-8

2.3 **The Visualization Effort** 2-10

2.4 **Real Time Computer Graphics** 2-11

2.5 **Hidden-Surface Removal Algorithms** 2-15

2.5.1 Object-Space Algorithms 2-15

2.5.2 Image-Space Algorithms 2-15

2.5.3 Binary Space Partitioning (BSP) 2-16

2.5.4 Brute-force Method 2-16

2.6 **Conclusion** 2-17

III. **System Analysis and Design** 3-1

3.1 **Introduction** 3-1

3.2 **System Analysis** 3-1

3.3 **System Design** 3-2

3.4 **Control-Flow** 3-3

3.4.1 Scalar Z-buffer Design Language 3-3

3.4.2 Vector Z-buffer Design Language 3-4

3.5 **Edge and Scan-segment Interpolation Methods** 3-13

3.5.1 Scalar interpolation 3-14

3.5.2 Bucketized interpolation 3-15

Page

3.5.3 Non-bucketized interpolation 3-16

3.6 **Data Structures** 3-18

3.7 **Methodology** 3-21

3.8 **Summary** 3-22

IV. **Implementation and Results** 4-1

4.1 **Introduction** 4-1

4.2 **Timing Studies** 4-1

4.2.1 Introduction 4-1

4.2.2 Timings are Hardware Dependent 4-1

4.2.3 Cray-2 Clock Functions 4-3

4.2.4 Statistical Variation 4-4

4.2.5 Timing Caveats 4-4

4.2.6 How the Code was Decomposed for Timing 4-9

4.2.7 Timings for the F-16 4-11

4.2.8 Timings for the "DodecaIcosahederon" 4-14

4.2.9 Summary of "DodecaIcosahederon Timings' 4-15

4.3 **Heuristics for Vectorization** 4-15

4.3.1 Introduction 4-15

4.3.2 Advantages and Disadvantages of Vectorization . . . 4-15

4.3.3 Twenty Heuristics for Producing Vectorized Software 4-17

4.4 **High Level Design for a New Vectorized Z-buffer Algorithm** 4-27

4.5 **Conclusion** 4-29

V. **Conclusions** 5-1

5.1 **Introduction** 5-1

5.2 **Recommendation for Future Thesis Research** 5-1

5.3 **The Economics of Real-time Image Display** 5-3

Page

5.4 **Conclusion** 5-4

Appendix A. **Pascal Vector Syntax Overview** A-1

Bibliography BIB-1

Vita VITA-1

List of Figures

Figure		Page
2.1.	Cray-2 Supercomputer	2-2
2.2.	Cray-2 Memory Organization	2-4
2.3.	Ultra Frame Buffer with Display	2-12
2.4.	Diagram of Cray-2 with Ultra Frame Buffer	2-13
3.1.	Polygon and Polygon Decomposed into Scan-segments	3-5
3.2.	Polygon Decomposed into Four Trapezoids	3-5
3.3.	Trapezoid Data Structure	3-19
3.4.	Trapezoid Height Bucket Data Structure	3-20
3.5.	Scan-segment Length Bucket Data Structure	3-21
4.1.	F-16 in the Normal View	4-5
4.2.	"DodecaIcosahederon in the Normal View	4-6
4.3.	Nested Record Data Structure	4-20
4.4.	Alternate Trapezoid Height Bucket Data Structure	4-21

List of Tables

Table		Page
2.1.	Performance Figures for Different Strides	2-5
4.1.	F-16 Polygon Statistics	4-7
4.2.	F-16 Trapezoid Statistics	4-7
4.3.	"DodecaIcosahederon" Polygon Statistics	4-8
4.4.	"DodecaIcosahederon" Trapezoid Statistics	4-8
4.5.	Speedup Table for F-16 Display	4-12
4.6.	Speedup Table for "DodecaIcosahederon"	4-15
4.7.	Pascal and FORTRAN Interpolation Timings	4-29

Abstract

Recent developments in scientific computing have prompted the need for supercomputer graphics research. These developments include the requirement to visualize large amounts of data which are processed by the supercomputer, preferably by real-time image generation. Unfortunately, most currently used graphics algorithms are not optimized for vector computers. This thesis involves the design and implementation of a hidden-surface removal algorithm for the Cray-2 vector supercomputer, with the goal of real-time image display.

A z-buffer hidden-surface algorithm, written in Pascal, was vectorized and implemented on the Cray-2. Special attention was directed toward the methodology of algorithm and data structure design to exploit the Cray-2 architecture. Timing studies comparing the vector version to the equivalent scalar version showed that while the Pascal vector code produced localized speedups, the vector code was less efficient than the scalar code. When critical portions of the code were translated from Pascal to FORTRAN, significant speedup was achieved, indicating that the FORTRAN compiler generates more efficient object code than the Pascal compiler. However, real-time performance was not achieved. Based on the knowledge gained from vectorizing the z-buffer algorithm, vectorization heuristics were discovered and a new algorithm which should provide further speedup is presented.

Experience from the thesis research indicates that when attempting to achieve real-time performance, consideration must be given to compiler efficiency, supercomputer architecture, type of data displayed, and display algorithm. Relying only upon good algorithm design may not produce adequate performance, and if any of these four areas is not properly addressed, performance suffers. Another important observation is that vectorization is difficult and can obscure the clear function of software, decreasing software readability and perhaps maintainability. Thus, transforming scalar software into software structured for vector operations should be done sparingly; only when absolutely essential to meet performance requirements. However, despite its difficulties, vectorization is a powerful way to increase software efficiency.

A VECTORIZED HIDDEN-SURFACE ALGORITHM IMPLEMENTED ON THE CRAY-2 SUPERCOMPUTER

I. Introduction

1.1 Introduction

Recent developments in scientific computing have prompted the need for supercomputer graphics research. These developments include the requirement to visualize the large quantities of scientific data which are processed by the supercomputer. Unfortunately, the lack of computer graphics algorithms specifically developed for the supercomputer architecture hinders this visualization. This thesis involves the design and implementation of a graphics algorithm developed specifically for the Cray-2 supercomputer architecture.

This chapter presents the background material which justifies the need for the thesis. It also examines the role of computer graphics workstations and their relationship to the supercomputer and discusses the probable contribution of the supercomputer in future computer graphics work. Furthermore, the assumptions necessary for completion of the thesis, as well as the scope of the thesis are addressed. Finally, the methodology and equipment needed to implement the thesis are discussed. As for the remainder of the thesis, Chapter 2 reviews the literature applicable to this thesis topic, and Chapter 3 presents the graphics algorithm design. The results are provided in Chapter 4 and Chapter 5 outlines the conclusions and recommendations for future research.

1.2 Background

1.2.1 The Popularity of Graphics Workstations Computer graphics, once confined largely to mainframe computers, is now supported by the small graphics workstation, primarily because modern integrated circuit technology can now support computationally intensive computer graphics algorithms by embedding the graphics algorithms in the

hardware. Such specialized hardware supports matrix transformations, hidden-surface removal, the display of polygons and bicubic surfaces, and a variety of other graphics display functions. Processors, memory, and the bus structure are designed and organized into an architecture which provides for efficient display of image data. These special purpose circuits are combined and packaged into computer graphics workstations, revolutionizing the image rendering process. With advances in Very Large Scale Integrated Circuitry (VLSI), graphics workstations can be expected to become more popular and less costly, while providing more capability. As a result of this technology, many computer scientists believe that the graphics workstation will usurp the role of conventional mainframes for processing and displaying graphics data (Frenkel, 1988:114). However, the memory and processor power required to handle large quantities of data typically generated by supercomputers can quickly overwhelm these graphics workstations.

1.2.2 The Need to Visualize Data Produced by Supercomputers A supercomputer is currently defined as a computer capable of 100 Million or more Floating Point OPerations per Second (FLOPS). However, as technology produces more capable computers, the definition of the supercomputer will change, but generally, a supercomputer provides a magnitude of computational power above that of nearly all other computers. Typically, supercomputers support vector operations and have several processors and specialized hardware such as multiple functional units and pipelines. Computational fluid flow analyses, weather simulations, image processing, nuclear effects simulations, and other time consuming applications are routinely performed on supercomputers and require memory and computational power that presently only supercomputers can provide (Staudhammer, 1987:25) and (Frenkel, 1988:119). Furthermore, these applications generate mountains of data much of which is provided as rows and columns of numbers, a format which is not efficiently comprehended by humans. Additionally, so much data is produced that it cannot be used (McCormick and others, 1987:vii,B-15).

Due to the magnitude of data being generated by supercomputers, an initiative in computing, termed "Visualization in Scientific Computing," is receiving attention from scientists in a variety of disciplines. Briefly stated, the goal of visualization is "to lever-

age existing scientific methods by providing new scientific insight through visual methods (McCormick and others, 1987:3)." The initiative is an attempt to transform numbers and symbols into images. Scientists can visually observe the results of simulations and computations by transforming collections of numerical data from a multitude of sources to visual data. This transformation into a visual medium recognizes that humans are capable of understanding and interpreting a vast amount of data when the data is presented visually in three dimensions. Those involved in the visualization effort believe that "the ability of scientists to visualize complex computations and simulations is absolutely essential to ensure the integrity of analyses, to provoke insight and to communicate those insights with others (McCormick and others, 1987:7)." Unfortunately, this data is so voluminous that using computer graphics workstations to reduce it to a visual form is costly and laborious and may be more quickly performed by supercomputers. Additionally, the volume of supercomputer data exceeds what can be transmitted to the workstation in any reasonable time using existing conventional networks (10-50 megabits/second).

1.2.3 The Future of Computer Graphics Debate on the future of computer graphics continues. While the need to visually display complex data generated by supercomputers is acknowledged, the precise mechanism for this display is unknown. The view of those who advocate using workstations to render and display the data is given by Bruce H. McCormick, one of the editors of a July 1987 report, "Visualization in Scientific Computing:"

> The visualization problem is not peculiar to supercomputers in any sense. In fact, I don't expect the bulk of [the visualization initiative] to be tied to them. Supercomputers are in many ways dinosaurs, a dying breed. Their costs indicate that that market will not be very significant within the computer market. The workstation market is much larger, and eventually number crunching and modeling will be delegated between supercomputers and workstations. Parallel processing will find its way into minisupercomputers, and as more and more firepower is needed they will also serve as local graphics and image processing machines (Frenkel, 1988:113).

Further elaborating the workstation position, Donald P. Greenberg, Professor of Computer Graphics at Cornell University, predicted that: "the future of visualization is going to depend on high-powered workstations, which are connected over high bandwidths

to supercomputers (Frenkel, 1988:114)." While most generally agree that the supercomputer is necessary for preliminary data collection and reduction, some believe it is too expensive to use supercomputers to interactively render the data. However, Nelson Max, a computer scientist at Lawrence Livermore National Laboratory disagrees: "as an algorithm developer, I cannot use a workstation that has specific rendering algorithms already built into it. I want to create new algorithms, and to do this, I need a general-purpose computer rather than specialized hardware to speed up old algorithms (Frenkel, 1988:119)." Nelson Max acknowledges that supercomputers are a costly means of doing business, but believes that for some problems there may be no alternative. And those who would delegate most computer graphics processing to workstations admit that "supercomputers are required when rendering software is not supported by workstation firmware or when pushing the frontiers of the number of floating point operations carried out per pixel (McCormick and others, 1987:B-13)." Certainly, the visualization of complex data pushes the state of the art in current image generation capability.

The entire debate is similar to the ongoing squabble between the advocates of sequential uniprocessor machines and the advocates of multiprocessor machines. Each group predicts the demise and extinction of the other. One is also reminded of the current view held by some computer scientists of the eventual demise of the mainframe computer, to be replaced by many smaller microcomputers.

1.2.4 Supercomputer Graphics Research Partially due to the high costs of supercomputing, current research in the area of developing supercomputer graphics algorithms is sporadic, but is gaining attention. Surprisingly, a major user of supercomputer graphics capability is the motion picture industry, for which the supercomputer is essential for creating animation (Frenkel, 1988:119). However, the supercomputer user base is small and software for supercomputers is expensive to develop, so the variety of software available to supercomputer graphics users has suffered and is less than that available to graphics workstation users. Additionally, supercomputer visualization software is "primitive, nonexistent, or very expensive," and "existing tools are not adequate to meet user needs (McCormick and others, 1987:B-15, ix)." Indeed, the supercomputer users are finding that

turing of computer graphics algorithms for the supercomputer is not productive. Third, hidden-surface removal algorithms are the most versatile and useful graphics algorithms for vectorization. Therefore, the search for algorithms to vectorize is limited to hidden-surface removal algorithms.

1.5 Scope

The scope of the thesis, which follows from the above assumptions, is to examine the supercomputing aspect of visualization. No attempt to restructure graphics algorithms for use on graphics workstations is made. This thesis concentrates on the performance of the scalar algorithm verses the vectorized algorithm but does not compare the value of the supercomputer to the graphics workstation for real-time graphics display. Furthermore, only hidden-surface removal algorithms are examined for vectorization, with the primary objective of gaining experience and insight into how to vectorize software. Thus, real-time performance is not critical for success, but is a goal.

The overall problem involves the development of a hidden-surface removal algorithm which is vectorized for the Cray-2 supercomputer. This enlarges the base of graphics software available to Cray-2 supercomputers users and provides a means for the visualization of some of the massive amounts of data generated by the Cray-2 supercomputer. The stopping criteria is the vectorization of one hidden-surface algorithm for the rendering of polygonal data (objects described solely by planar polygons) and bicubic surfaces (objects described by bicubic patches).

1.6 Approach/Methodology

The approach to the required research is:

1. Examine the Cray-2 supercomputer architecture to determine how an algorithm is best restructured to take advantage of the Cray-2 hardware. For the successful vectorization of software, the supercomputer hardware must be understood.
2. Review the literature for vector and parallel algorithms. Special attention is given to identifying features of an algorithm which make it a good candidate for vectorization.

3. Review and survey existing image rendering hidden-surface algorithms for polygonal surfaces. Among the algorithms that may be considered are: Z-buffer, A-buffer, Binary Space Partitioning, Depth-sort, and Scan-line algorithms.

4. Review and survey existing bicubic surface image rendering hidden-surface algorithms. Among the algorithms that may be considered are: the Blinn algorithm, the Whitted algorithm, the Lane-Carpenter algorithm, and the adaptive forward differencing algorithm.

5. Choose one of the algorithms in the above two categories that provides the greatest potential for vectorization and implement it on the Cray-2 supercomputer. Special attention is directed toward the methodology of algorithm and data structure design to exploit the supercomputer hardware.

6. Conduct timing studies on the Cray-2 supercomputer. The ability to achieve near real-time animation (10 frames per second) of reasonable object descriptions (1000 to 10,000 polygons) is targeted. A comparison of the Cray-2 scalar algorithm execution time to the vectorized algorithm is performed.

7. Collect heuristics for successful vectorization based upon experience developing software for the Cray-2.

1.7 Materials/Equipment

The Air Force Weapons Laboratory (AFWL) at Kirtland Air Force Base, New Mexico, provides access to the Cray-2 supercomputer with a Pascal vectorizing compiler for implementing the hidden-surface removal algorithm. The Cray-2 is connected by a 100 megabyte per second channel to an Ultra frame buffer with 1280 x 1024 pixel resolution display. For work done at the Air Force Institute of Technology, an Arpanet connection to the facilities at AFWL is provided.

1.8 Conclusion

This work provides insight into how algorithms, particularly computer graphics algorithms, can be vectorized to exploit supercomputer architectures. The strengths and

weaknesses of the supercomputer for graphics work are revealed. Additionally, it reinforces the usefulness of the supercomputer for displaying data and contributes to the supercomputer software base. Finally, the supercomputer may provide adequate speedup to show that images can be rendered in real-time.

II. Literature Search

2.1 Introduction

Chapter 2 reviews the background information necessary to investigate the practicality of using the supercomputer to render images in real-time by restructuring computer graphics algorithms to exploit the advantages of vector and parallel computers. Because computer graphics algorithms are computationally intensive, they may benefit from the speedup achieved by vectorizing the algorithm for execution on the Cray-2 supercomputer. Successful vectorization requires a working knowledge of the computer architecture and this thesis discusses the Cray-2 supercomputer architecture and some of its features which provide for the speedup of conventional computer graphics algorithms. The visualization of scientific data, and its relationship to the display of real-time computer generated images is also examined. Finally, the algorithms for displaying images and the general qualities of algorithms which provide the greatest potential for vectorization are identified.

2.2 The Cray-2 Supercomputer

2.2.1 Architecture The Cray-2 supercomputer is the successor to the Cray-1 and is a Multiple Single Instruction Multiple Data (MSIMD) vector computer. Its architecture is similar to the Cray-1, but it has been improved. Operating at a 4.1 ns clock rate, the Cray-2's four processors can provide peak performance of over 975 Million Floating Point OPerations per Second (MFLOPS). The Cray-2 boasts 256 million 64-bit words of main memory (2 gigabytes). It contains eight 32-bit address registers (A0-A7), eight 64-bit scalar registers (S0-S7), and eight-64 word vector registers (V0-V7, 4096 bits per vector register). Each of its four processors contains nine functional units to perform address add, address multiply, scalar integer, scalar shift, scalar logical, vector integer, vector logical, floating point add, and floating point multiply operations (Cray Research, 1987:2-8). It supports multitasking and multiprogramming. The Cray-2 supercomputer with four processors can support a maximum of eight HSX controllers, each providing an 853 megabits per second connection to high-speed devices. This large bandwidth is absolutely essential for real-time display. Figure 2.1 shows the Cray-2 supercomputer.

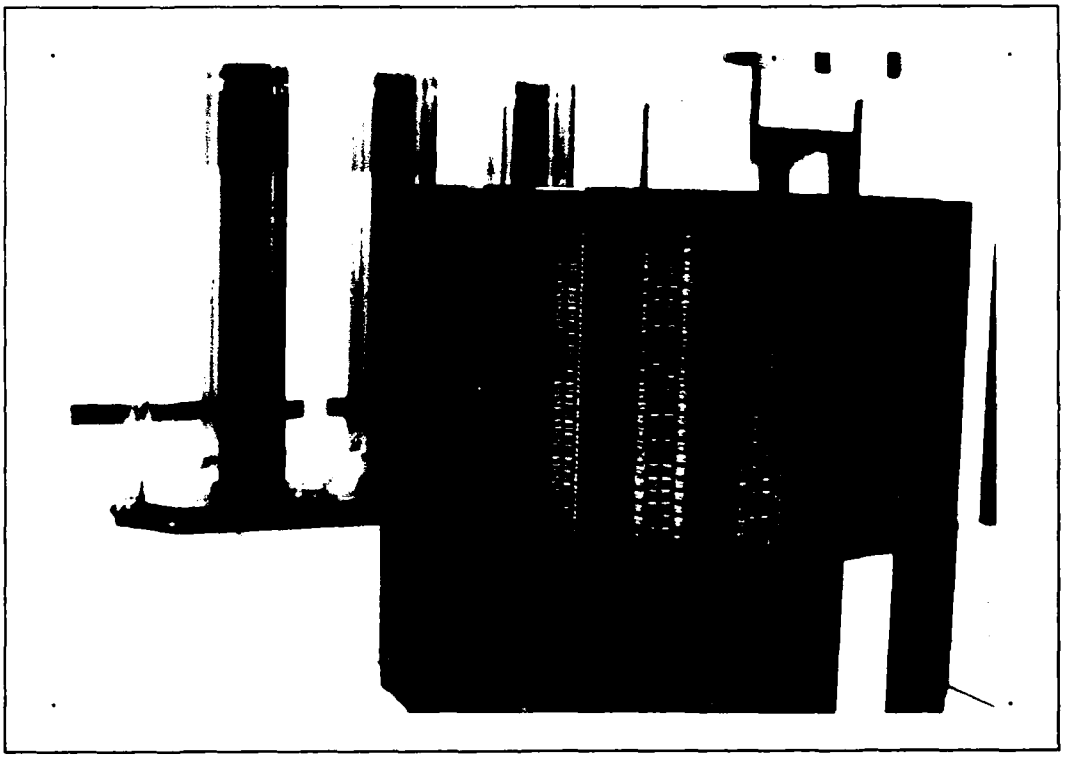


Figure 2.1. Cray-2 Supercomputer

The clear tubes behind the cylindrically shaped computer are reservoirs used to hold the inert fluorocarbon liquid when the Cray-2 is drained for maintenance. This inert fluorocarbon liquid engulfs and cools the Cray-2's integrated circuitry.

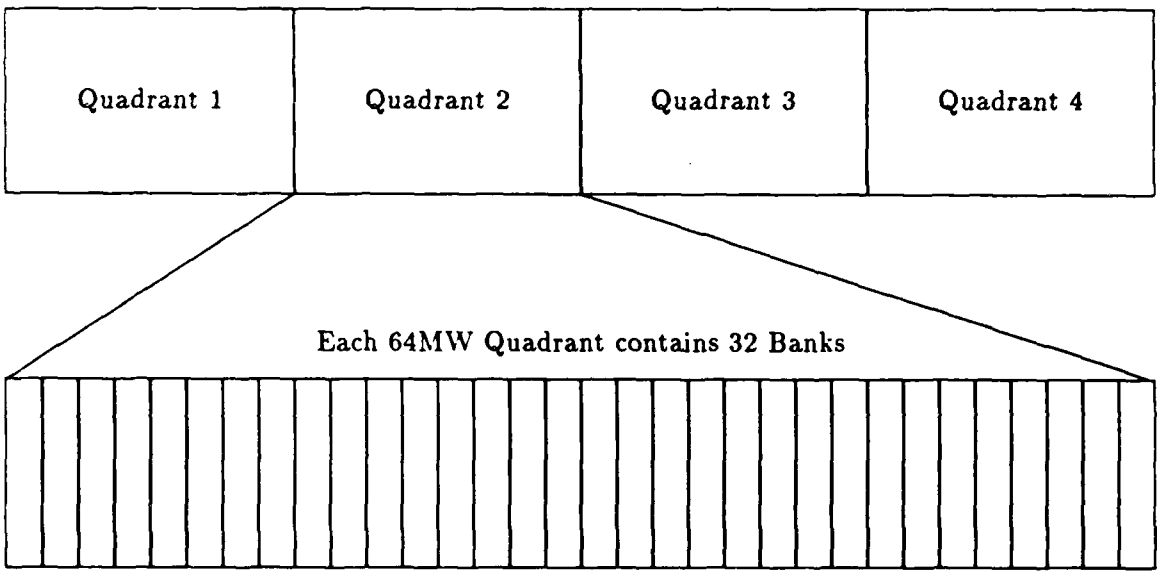
2.2.2 Cray-2 Memory Organization An understanding of the Cray-2's memory organization is essential for effective vectorization. Due to the Cray's large memory capacity, program execution speed can be significantly affected by haphazard memory reference. Figure 2.2 depicts the memory organization.

The Cray-2's 256 MW of memory is dynamic metallic oxide semiconductor (MOS). The memory is divided into four quadrants of 64M words each, each quadrant containing 32 2MW banks, and each bank is subdivided into two pseudo-banks (Cray Research, 1987:B-3).

According to the Air Force SuperComputer Center, access to the Cray-2's memory is "considerably slower than access to the smaller (2MW) memory of the Cray-1 (Kamrath,1988:4)." The Cray-2's four processors have access to any given quadrant every fourth clock period. During the next three clock periods, a processor is prevented from accessing the quadrant it most recently accessed. Thus, if a computer scientist designs a data structure in which elements that occupy the same quadrant are referenced in sequence, the memory bottleneck degrades performance appreciably and may eliminate or reverse any speedups due to vectorization. The way in which elements in memory are accessed is referred to as stride. For example, a stride of one denotes the accessing of successive words in contiguous memory, while a stride of two denotes accessing elements in every other word of memory. Table 2.1 lists the Cray-2's performance, in MFLOPS, for some simulations in which the stride was varied. Note the serious degradation in performance as the stride varies from one to 256.

Avoiding access of the same quadrant in succession is not the only concern the user faces. Further attention must be directed to avoiding access of the same pseudo-bank in succession. After a pseudo-bank is referenced, it cannot be referenced again until after the chip reservation time has expired. The chip reservation time is 45 clock periods. Thus, if the same pseudo-bank is accessed twice in a row, the processor must idle for 45 clock periods

256 MW of Main Memory is divided into 4 Quadrants



Each 2MW Bank contains 2 Pseudo Banks

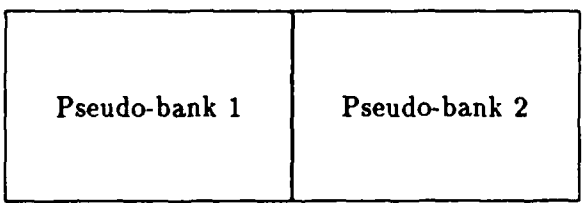


Figure 2.2. Cray-2 Memory Organization (Conley, July 1988)

Stride	MFLOPS
1	83.33
2	31.25
3	83.02
4	19.23
5	82.54
32	4.43
64	1.50
128	0.87
256	0.50

Table 2.1. Performance Figures for Different Strides (Kamrath,1988:4)

after the first access. If different pseudo-banks in the same bank are accessed in succession, the processor idles 25 clock periods between access. It should now be evident that data structure design and control structure design have a profound impact on performance. For maximum performance, the user must be careful to ensure that his vector operands occupy contiguous elements in memory, with the optimum stride of one. Strides that are non-zero powers of two must be avoided, and strides of 256 are the worst.

2.2.3 The Advantages of Vectorization The primary method that the Cray-2 uses to increase its speed over conventional computers is its special design which has been optimized to perform vector operations. In *High-Performance Computer Architecture*, Stone lists four advantages of vector computers:

1. "Provide vector instructions to take advantage of [vectorization] for numerical applications.
2. Provide facilities to extend the range of applicability of the architecture beyond vector processing.
3. Use multiple levels of memory, particularly high-speed buffers; and
4. Mix pipeline and parallel techniques in various degrees to achieve an acceptable value to price performance (Stone, 1987:271)."

2.2.4 The Advantages of Pipelining Vector operations are implemented using vector pipelines, enabling vector computers to produce one arithmetic computation (an add,

multiply, or divide) at a maximum of one result every clock cycle. A pipeline is a piece of computer hardware which breaks an operation into a number of tasks, each of these tasks executing simultaneously with other tasks. Stone defines a pipeline as "a structure that consists of a sequence of stages through which a computation flows with the property that new operations can be initiated at the start of the pipeline while other operations are in progress through the pipeline (Stone,1987:406)." Pipelines enhance computer performance through temporal (time) parallelism, while parallel processors achieve their improvements through spatial (space) parallelism. Both are forms of parallelism, and the Cray-2 uses both to achieve performance improvements.

A pipeline may contain any number of stages, each stage a subset of the particular operation to be performed. A pipeline is analogous to an assembly line, and for computer purposes, it is basically an arithmetic operation assembly line. For example, a floating point multiply pipeline might contain three stages, each performing a part of the floating point multiply operation in sequence:

1. Add the exponents,
2. Multiply the mantissas, and
3. Normalize the result.

This pipeline would produce a maximum speedup of three over a non-pipelined approach to floating point multiply. Generally, for a vector pipeline with K stages, it takes K clock periods for the first vector element to emerge from the pipeline, and each additional vector element takes one clock period. However, for a scalar operation, if the first element of a vector takes N clock periods to complete, then each additional element takes N clock periods to complete. If the scalar time to operate on the first vector element is the same as the vector time to operate on the first vector element ($K = N$), then theoretically, a pipeline with K stages can produce a speedup of K times over a non-pipelined operation, assuming the filling and emptying of the pipeline is negligible. Pipelining also provides for the efficient use of memory bandwidth (Stone, 1987:274), absolutely necessary for supercomputers with large memory capacity.

An algorithm which operates efficiently in scalar may not be the best candidate for vectorization. This is due to the overhead required to set up the vector pipeline. In general, the breakeven point varies depending on the operation, according to the equation:

$$T = V(t) + L - 1 + O \tag{2.1}$$

where T is the total time to implement a vector operation, $V(t)$ is the time to implement the first vector operation, L is the vector length, and O is the additional overhead required to set up the vector pipeline over the scalar operation. The breakeven point is achieved when:

$$L \times S(t) \geq T \tag{2.2}$$

where $S(t)$ is the time to implement one scalar operation. Substituting the expression for T given by Equation 2.1 into Equation 2.2 results in:

$$L \times S(t) \geq V(t) + L - 1 + O \tag{2.3}$$

If the overhead is negligible (0 clocks) and V(t) = S(t), then it is advantageous to vectorize when the vector length is one or more. (On the Cray-2, V(t) is always slightly greater than S(t)). However, if the overhead is equal to the time to execute one operation, then the vector length must be greater than two before the breakeven point is achieved. Thus, one must not strive to vectorize everything, but only those areas where speedup is achieved over the scalar operation. An important caveat must be observed. Only hardware execution speeds have been considered here, but software execution is an equal if not a more important consideration. The time to set up a loop that is executed in scalar mode may be more time consuming than the vector implementation of the loop, even though the vector length is smaller than the required breakeven vector length, because the software loop overhead may cause the vector operation to execute more efficiently (Conley, June 1988). Conversely, the vector implementation of a program may be less efficient than the scalar version because of the software overhead associated with manipulating data to take full advantage of vector operations.

Unfortunately, pipelining is an expensive method to produce results when the pipeline must be prematurely emptied for any number of reasons. Branching instructions and

pipeline interlock (caused by recurrence relations) can significantly degrade the performance of a pipeline (Stone, 1987:148). With the overhead to set up a pipeline, it is possible that a pipeline can cause the computer to perform more poorly than if there had been no pipeline. There are methods to minimize these problems, such as branch prediction techniques, but the most efficient method for solving the problems posed by pipelines is to structure the software in a manner that branches and interlocking are minimized.

Another requirement is to minimize the use of operations which are time-consuming. On the Cray-2, division is more expensive than multiplication, and division must be avoided whenever possible. The Cray-2 has no hardware divider, so it must compute the reciprocal of the divisor and multiply by the reciprocal. However, forming this reciprocal is a time consuming process, and if the same divisor is used repeatedly in a number of statements, it is a good programming practice to place the reciprocal of the divisor in a temporary variable and multiply by the temporary variable.

2.2.5 Vectorizing Compilers Vectorizing compilers automatically structure software to take advantage of pipelined and vector architectures. While vectorizing compilers can decrease the inefficient use of the pipeline, the software must still be structured in a manner that minimizes pipeline conflicts. Hierarchical data structures that make extensive use of pointers, and the use of recurrence relations and recursive routines can be particularly hazardous to efficient pipeline operation. (Dyer and Whitman, 1987:38), (Higbie, 1983:180), and (Buzbee, 1986:191). Thus, any software that attempts to take advantage of vector operations should minimize their use. A number of vectorizing compilers are available for the Cray-2: Pascal, FORTRAN, and C.

2.2.6 Graphics Supercomputers A promising development in computer graphics is the realization of the advantages that vector supercomputers provide for computer graphics users. Two computer manufacturers have released their first version of graphics minisupercomputers. Stellar has unveiled its GS1000 graphics minisupercomputer, and a competitor, Ardent, released its Titan minisupercomputer. Both were introduced in March 1988. Also included is a discussion of a superworkstation, the Silicon Graphics IRIS GTX. While these minisupercomputers and superworkstations cannot match the Cray-2's performance in raw

computing power, their reported performance for rendering graphics images is astounding. Keep in mind, however, that the graphics minisupercomputers' performance is achieved by implementing the most time consuming operations in specialized graphics hardware, while the Cray-2 has no such specialized hardware and all graphics functions must be implemented in software.

2.2.6.1 Stellar GS1000 Computer Stellar's machine is a 20 to 25 Million Instruction per Second (MIPS) machine with four vector pipelines, capable of a peak performance of 40 MFLOPS. Its memory capacity is 128 MB, and it can display 150,000 Gouraud shaded z-buffered triangles per second (Apgar,1988:255). Its speed at rendering triangles is derived from the combination of a special purpose graphics engine which determines pixel color values and a general purpose vector floating point processor pipeline for performing coordinate transformation, clipping, matrix operations, and other functions which map easily to a pipeline architecture (Apgar,1988:258).

2.2.6.2 Ardent Titan Ardent released a computer similar in design to the Stellar GS1000. Its clock operates at 62.5 ns and it may contain up to four 32 bit, 16 MIP R2000 processors and 128 MB of memory (Diede and others, 1988:14). Its peak performance is 64 MFLOPS. Its architecture features three functional units for floating point arithmetic, multiplication, and division. Like the Stellar GS1000, it uses vector pipeline units for object coordinate tranformation, perspective division, clipping, and lighting calculations, and it contains pixel and polygon processors responsible for determining pixel color values (Diede and others, 1988:27). Capable of displaying 200,000 Gouraud shaded polygons per second, it supports animation rates of 10,000 Gouraud shaded triangles at 15 frames per second (Ardent, 1988).

2.2.6.3 Silicon Graphics IRIS GTX Silicon Graphics recently released their IRIS GTX "superworkstation." While Silicon Graphics calls it a superworkstation and not a minisupercomputer, its performance nearly matches that of the Stellar and Ardent machines, and it boasts many of the same architectural features. Its CPU includes two 16.7 MHZ R2000 processors with two floating point coprocessors, and up to 128 MB of

memory. Capable of 20 MIPS, the graphics system can render 100,000 z-buffered, Gouraud shaded, 10 x 10 pixel quadrilaterals per second (or 120,000 triangles per second) (Akeley, 1988:239). The graphics system is composed of four subsystems:

1. Geometry subsystem: responsible for transforming world to eye coordinates, clipping, and other coordinate transformation functions. This subsystem features five geometry engines pipelined together, each capable of 20 MFLOPS (100 MFLOPS total). Each geometry engine is a six stage pipeline (Akeley, 1988:241,243).
2. Scan-conversion subsystem: responsible for edge and scan-segment interpolation. Edge and scan-segment interpolation are described in Chapter 3.
3. Raster subsystem: maintains the 1280 x 1024 pixel frame buffer (Akeley, 1988:241).
4. Display subsystem: supplies video information to the display (Akeley, 1988:241).

All three computers share a similar partitioning of the hidden-surface display problem. Vector pipelines are used for coordinate transformations, perspective division, lighting calculations, and clipping, while edge and scan-segment interpolation and display are handled by special purpose graphics hardware.

2.3 The Visualization Effort

Supercomputers produce incredible amounts of data describing a variety of natural and man-made processes. The data is frequently produced in the form of numbers and text, and transforming this data into a visual format enhances the efficiency with which humans can understand and interpret the data. However, many supercomputers do not have a graphics environment to ease the manipulation and display of the data. Typically, this data requires a great deal of memory, and downloading it to graphics workstations over low bandwidths is time-consuming and expensive. Much of this data is collected from sensors monitoring dynamic processes. For example, the sensors monitoring the change in airflow over an aircraft wing can quickly produce numerical data beyond the ability to comprehend it. Certainly the understanding of this airflow data is enhanced if transformed and displayed in real-time. Visualizing this data in real-time can provide scientists with

additional insight that would not be possible if they had to rely solely on the numerical data (Special Report, 1987:65).

2.4 Real Time Computer Graphics

Visualizing data in real-time is a difficult task. Many computer generated images are static images. These images depict motionless objects, and for a very good reason – motion display is an expensive process. Yet, for the full impact of the visualization of computer data to occur, these images must show motion, for the impact of motion on an audience is almost always more profound than static displays, and there are numerous problems, such as fluid flow analysis, which are best visualized in real-time.

Real-time motion is produced by displaying images in a sequence approximately 30 to 60 times per second (Staudhammer, 1986:296). Near real-time motion is produced when the images are displayed at rates around 10 times per second. For the best effect, a display rate of 60 times per second is desirable as an annoying and distracting flicker occurs when images are displayed at less than this rate.

As technology progresses, and as processors become faster, this real-time update rate becomes more feasible for the same size raster display. However, technological innovation is also producing less costly, higher resolution raster displays, complicating the ability to display images in real-time. For a raster display with a 500 x 500 pixel resolution, a 30 hertz update rate corresponds to 133.2 ns update time per pixel. A 1000 x 1000 pixel CRT resolution requires a 33.3 ns access time per pixel for the same display rate. Thus, although the perimeter of the display is doubling, the update rate required quadruples, leading to an inverse relationship between display refresh rate capability and display resolution. Either the viewer has to be content with lower resolution or a slower refresh. The Ultra frame buffer, which displays images generated by the Cray-2 supercomputer, supports a 1280 x 1024 pixel resolution. The Ultra is double buffered, supporting two 2048 x 1024 pixel buffers which allow one screen to be displayed while the other is loaded. It can display a 1280 x 1024 image at 14.7 frames per second, about half the real-time rate, when configured with buffer switch at vertical retrace. The Ultra frame buffer with display is

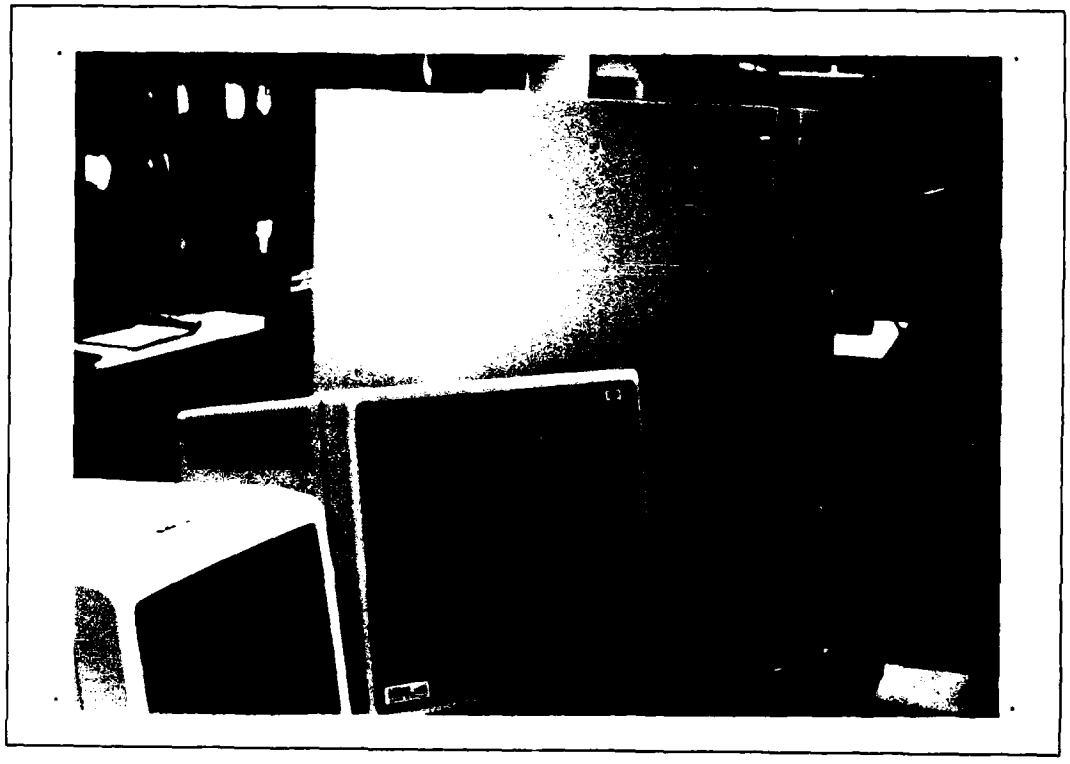


Figure 2.3. Ultra Frame Buffer with Display

shown in Figure 2.3. The Cray-2 and Ultra frame buffer block diagram are shown in Figure 2.4.

In order to achieve real-time performance, the graphics software must produce the color value for all pixels on the display in one-thirtieth of a second, and consequently, the algorithm chosen to display the objects must be fast. Most graphics software is computationally intensive and for real-time display requires computational power greater than that available from most computer graphics workstations. (Keep in mind that we are discussing graphics software, not graphics algorithms embedded in hardware). Indeed, Staudhammer implies that, at least for the present, "the computations required for detailed image presentation seem to mandate supercomputer usage. Currently-used algorithms are poorly suited to supercomputer's vectorization requirements. A critical view needs to be taken at re-casting the display calculations into forms more adaptable to vector machines (Staudhammer, 1986:297)." Dyer and Whitman designed vectorized z-buffer software, but its performance is not real-time (Dyer and Whitman, 1987).

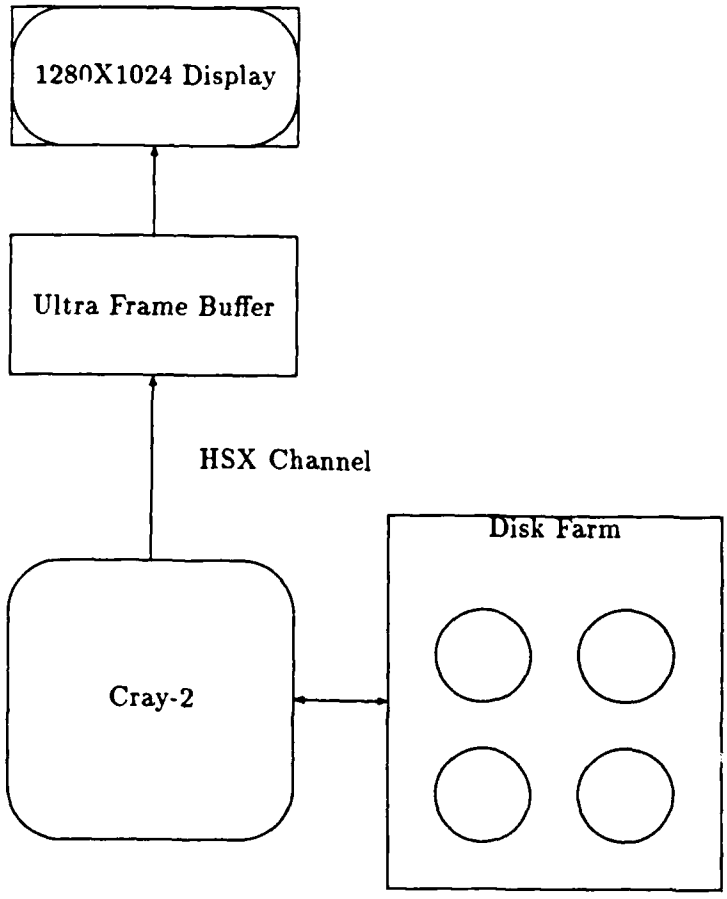


Figure 2.4. Diagram of Cray-2 with Ultra Frame Buffer

A powerful computer is not all that is required. Once the computational needs are met, the bottleneck may shift to the input/output capability of the display device. For real-time graphics, the display must be connected to the computer generating the pixel values by a high bandwidth connection capable of roughly 30 hertz × 1280 × 1024 pixels × 24 bits/pixel = 944 Mbits/sec, an incredibly high bandwidth. Such a bandwidth is not easily achieved, but the bandwidth requirement can be reduced using data compaction and other techniques. Of course, data compaction requires additional, scarce time. The HSX Controller, providing an 853 megabit per second bandwidth to the Ultra frame buffer, supports a maximum Ultra frame buffer display rate of 20.34 frames per second, assuming 32 bits per pixel must be transmitted. Of course, this display rate exceeds the ability of the Ultra when buffer switch occurs at vertical retrace, and thus the display and not the HSX connection is the bottleneck.

There are several ways to display objects in real-time. One of these ways is to render each frame of a scene individually and display each frame in non-real-time. Each frame is stored in memory, displayed, photographed, and then a motion picture of the display can be replayed in real-time by a movie projector or videotape. This is how the motion in some science fiction movies is synthesized, but it is extremely time consuming and in many cases is not feasible for the viewing of scientific data. Another method involves storing a sequence of frames in the computer's memory and displaying the frames as fast as the display can be driven. The drawback to this approach is the large amount of memory required to hold the frames prior to display and the requirement for a high bandwidth connection from the computer memory to the display. The most flexible, yet difficult approach is to render and display the image in real-time.

By now, it is evident that real-time image generation is a particularly time consuming process. However, to gain the advantages of real-time visualization, the need for quick display must balance with the need for an accurate image. Staudhammer noted that a simple light reflection model and crude surface approximations are usually adequate for rendering objects in real-time (Staudhammer, 1986:296). Some of these surface approximation algorithms can be classified as hidden-surface removal algorithms. These hidden-surface algorithms can be combined with a simple lighting model and if the algorithm produces results

sufficiently fast, real-time performance is achieved. In real-time display, the most commonly used lighting reflection model is Gouraud shading (Bishop and Weimer, 1986:103). While flat shading is less computationally intensive than Gouraud shading (and quicker), the extra realism that Gouraud shading provides is usually worthwhile.

2.5 Hidden-Surface Removal Algorithms

There are numerous algorithms used to display graphics data. They vary in complexity and speed. Any algorithm used to display real-time data should be relatively simple and have a hidden-surface removal capability. The goal of hidden-surface removal is to determine which surfaces of a given object are not visible from a particular viewpoint and to display only those surfaces which are visible. Hidden-surface removal is a fundamental requirement for most images.

There are several methods to achieve hidden-surface removal. In "A Characterization of Ten Hidden-Surface Algorithms" (Sutherland and others, 1974), most of the hidden-surface removal algorithms mentioned fall into one of two categories: object-space algorithms and image-space algorithms.

2.5.1 Object-Space Algorithms Object-space algorithms determine which surface is closest to the viewer by comparing each surface to all other remaining surfaces and eliminating surfaces or portions of surfaces which are invisible. The computational effort required is proportional to the square of the number of surfaces and these algorithms are generally more time consuming than image-space algorithms (Foley and Van Dam, 1982:553), and thus they are not further considered for vectorization.

2.5.2 Image-Space Algorithms Image-space algorithms determine which surface is visible at any particular resolution point (pixel) on the display. All the surfaces which compose a scene are examined at each resolution point to determine which surface is the closest to the viewer. The computational effort required is proportional to the product of the number of resolution points (usually pixels) and the number of surfaces (Foley and Van Dam, 1982:553). There are a number of image-space algorithms which are used to

render images, including depth-sorting and scan-line algorithms. Depth-sorting and scan-line algorithms have been rejected as candidates for vectorization because of the sorting requirements for these algorithms. As the number of objects in a scene increases, significant performance degradation occurs because of the non-linear nature of sorting. At best, any sort is $O(nlogn)$, and many are $O(n^2)$. Two algorithms which hold the greatest potential for real-time performance are Binary Space Partitioning and the z-buffer approach and are discussed in the following sections.

2.5.3 Binary Space Partitioning (BSP) BSP algorithms have been used successfully for displaying data in near real-time. BSP algorithms require the preprocessing of object descriptions in object-space and display of the objects in image-space. Such algorithms, while good for static environments in which only the user's viewpoint changes, do not perform especially well when the environment contains a large number of moving objects (Sutherland and others, 1974:24). BSP algorithms are usually recursive, which make them unsuitable candidates for vectorization. BSP algorithms can be restructured to make them iterative, but the BSP algorithm is naturally expressed recursively, and restructuring it iteratively generally makes it less readable. An undesirable feature of the BSP algorithm is the non-linear increase in processing time as the number of objects in the scene increases.

2.5.4 Brute-force Method Sutherland discusses a brute-force method to solving the hidden-surface problem (Sutherland, 1974:51). The brute-force method later became known as the z-buffer algorithm. At the time it was introduced, its use was limited due to high memory requirements. However, with the availability of cheap and highly dense memory chips, it has become a popular approach. Its advantages include simplicity and relatively constant performance, regardless of the number of polygons in a scene, because as the number of polygons increases the number of pixels covered by each polygon tends to decrease (Foley, 1982:561). At worst, its performance grows linearly with the number of polygons. Since visualization data typically involves the display of complex objects, an algorithm which performs well for a large number of objects is preferred. Among its disadvantages is susceptibility to aliasing because it processes polygons in an arbitrary order. Sutherland, in "A Characterization of Ten Hidden-Surface Algorithms," presents an

interesting comparison of hidden-surface algorithm performance in Table VI (Sutherland, 1974:54). In it, he shows the constant performance of the z-buffer approach as the scene complexity increases. All of the other algorithms compared against the z-buffer increase non-linearly with scene complexity, and for complex scenes, the z-buffer approach is usually most efficient. For relatively non-complex scenes, the other image-space algorithms perform better than the z-buffer approach, but as the number of objects in a scene increases, the z-buffer's superior performance is quite pronounced. Since the demand for higher quality images is increasing, not decreasing, an algorithm which performs linearly with scene complexity is essential. Thus, it is understandable why companies that specialize in computer graphics equipment, like Silicon Graphics, Stellar, and Ardent are using the z-buffer approach in their latest state of the art computers to attempt real-time image generation.

2.6 Conclusion

It appears that the most promising way to achieve real-time display of images is through speedy display architectures, connected by high bandwidths to display devices, coupled with a highly efficient, vectorized, hidden-surface removal algorithm. The following need to be considered when designing real-time hidden-surface removal algorithms:

1. Use a simple lighting model.
2. Do not attempt antialiasing or other expensive ways to enhance the image.
3. Connect the display to the rendering computer by a high bandwidth connection.
4. Exploit the architecture of the vector supercomputer to fullest extent.
5. Use a relatively simple display algorithm that is not recursive and minimizes the use of hierarchical data structures.

From initial research into hidden-surface algorithms, the z-buffer approach seems the best candidate for achieving real-time performance by vectorization.

III. System Analysis and Design

3.1 Introduction

This chapter covers the analysis and design of the system software which meet the research requirements as outlined in Chapter 1, approach and methodology section.

3.2 System Analysis

System analysis is the process of identifying what the system needs to do and Chapter 1 and Chapter 2 discussed the research into the development of a system for visually displaying data in real-time. The algorithm chosen for investigation and implementation, based on preliminary efficiency considerations and potential for vectorization, is the z-buffer depth algorithm. The z-buffer algorithm is a well known and widely used hidden-surface removal algorithm. It requires a large amount of memory, which limited its use in earlier generation computers. Memory consumption, however, is not an important consideration for the Cray-2, which supports two gigabytes of main memory.

The z-buffer derives its name from its requirement for a z-buffer (also known as a depth buffer) in which the closest polygon depth at each resolution point is stored. It also requires a refresh buffer which stores a red-green-blue (RGB) color triple for each point. The depth buffer is initialized to a background depth value, and the refresh buffer is initialized to a background color. Then polygons are processed in an arbitrary order and are scan-converted into the refresh buffer. Scan-conversion is the process of determining right and left polygon edge values for each particular scan-line which intersects a given polygon. (Scan-lines are analogous to the horizontal lines on a raster display). The right and left edge values define a scan-segment. For all the resolution points which a given scan-segment covers, depth and color values are computed. For each point inside the scan-segment, the depth value is compared to the current z-buffer depth value. If the depth of the currently examined point is closer to the viewer than the z-buffer depth, then the z-buffer depth at that point is updated with the currently examined resolution point depth, and the refresh buffer is updated with the RGB color of the pixel. This process continues until all the polygons are scan-converted. Usually, the resolution points are mapped one-

to-one onto the raster display pixels, but if the image is anti-aliased then several resolution points may map to one pixel.

Foley and Van Dam summarize the z-buffer algorithm with the following process description (Foley,1982:560): "during scan-conversion, the following steps are performed for each point (x,y) inside a polygon:

1. Calculate the polygon depth z(x,y) at (x,y);
2. If the depth, z(x,y), is less than the z-buffer depth at (x,y), then:
   - Place z(x,y) into the z-buffer at (x,y) and
   - Place the pixel value of the polygon at z(x,y) into the refresh buffer at (x,y)."

Note that the above algorithm may be implemented in a number of different ways and its scalar implementation may differ from its vector implementation. Mr. Bob Conley, a computer scientist at the Air Force Weapons Laboratory (AFWL), has already implemented a Pascal z-buffer algorithm that operates in scalar mode on the Cray-2. This thesis centers on restructuring and vectorizing this code.

3.3 System Design

System design identifies how the system fulfills the requirements. The scalar z-buffer software is designed in a top-down structured manner, and top-down design techniques are used for implementing any modifications to the scalar code. Using the Pascal programming language eases the implementation of a top-down structured design, due to Pascal's strong-typing, abundance of facilities for specifying user defined types, and its numerous control constructs. The Cray-2 Pascal compiler is an extension to the International Standards Organization (ISO) Pascal. Since ISO Pascal does not support a syntax which is easily mapped to the Cray's vector units, Cray-2 Pascal supplies extensions to ISO Pascal which allow the computer scientist to express vector syntax naturally.

The system design can be broken into two parts: the program's control flow and the required data structures. The following three sections describe the control flow for

the scalar z-buffer, and the alterations to the scalar z-buffer to make the code suitable for vectorization. Then, the interpolation methods and data structures necessary for the vector version are defined.

3.4 Control-Flow

3.4.1 Scalar Z-buffer Design Language Before vectorizing the scalar z-buffer, it is necessary to understand how the scalar version of the z-buffer program works. The high-level design language for the scalar z-buffer with flat shaded polygons works as follows:

1. Read the object and polygon descriptions.
2. Transform polygon local coordinates to world coordinates.
3. For all frames to be generated do #4 - #7 below.
4. Transform the input polygons to the desired view by rotating, scaling, or translating the polygons and autoscale, which centers the object on the screen.
5. Transform all polygon world coordinates to eye coordinates.
6. For each polygon do:
   (a) Clip the polygon to the screen.
   (b) Purge if a backfacing polygon, perform flat shading calculations and perform perspective division.
   (c) Each polygon is composed of a number of edges. These edges must be broken into a number of small horizontal segments, each segment corresponding to a pixel on the display. Determine the right and left edge pairs for each scan-line which intersects the polygon in screen- space, and calculate the depth values for all right and left edge pairs. This is done using a multiaxis interpolation routine based upon the interpolation method presented in "A Fast Shaded-Polygon Renderer" by Roger Swanson and Larry Thayer (Swanson and Thayer,1986). This step is *edge interpolation* and produces scan-segments.

(d) For each scan-segment, determine the depth value of all pixels composing the scan-segment. This is *scan-segment interpolation.* If the depth value for a pixel is less than the z-buffer depth value, deposit the pixel color into the refresh buffer and update the z-buffer depth value with the new depth value.

7. Display the contents of the refresh buffer on the Ultra display.

Reading the object and polygon descriptions requires reading from a disk file. This is a particularly time consuming portion of the code, and since it is only performed once regardless of the number of frames generated, it is not included in timing studies. The file format description is included with the source code and is available upon request.

The scalar program also includes a bicubic display algorithm. Once the bicubic surfaces are broken into triangles, the above z-buffer algorithm is used to render the triangles. The decomposition of the bicubic surfaces into triangles has not been vectorized. One of the factors which makes vectorization of the decomposition difficult is the recursive nature of the decomposition, since recursion defeats vectorization.

3.4.2 Vector Z-buffer Design Language Many of the ideas for vectorizing the z-buffer algorithm came from two excellent articles: "A Vectorized Scan-line Z-Buffer Rendering Algorithm," (Dyer and Whitman, 1987) and "Fast Scan-line Conversion Using Vectorisation," (Goldapp, 1986). These ideas, as well as ideas from Mr. Bob Conley (AFWL/SCPS), are incorporated into the algorithm for the vectorized z-buffer.

Any z-buffer algorithm performs two basic functions which can be structured for vectorization. These two functions involve polygon (or trapezoid) edge interpolation (which produces scan-segments) and scan-segment interpolation, which produces pixel color values. Figures 3.1 and 3.2 show a polygon decomposed into scan-segments and a polygon decomposed into trapezoids.

Another important calculation determines the shading of each polygon or trapezoid. There are three primary methods of shading: Flat, Gouraud, and Phong:

1. Flat shading. This is the only shading method vectorized, although the scalar version of the program implements all three shading methods. Flat shading requires the

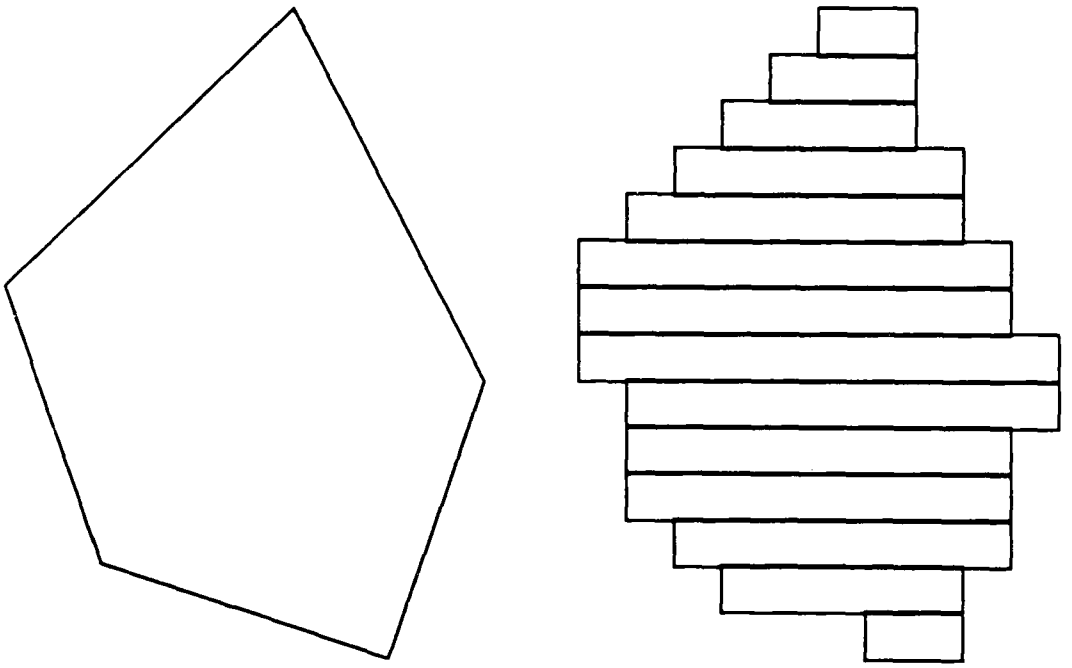


Figure 3.1. Polygon and Polygon Decomposed into Scan-segments

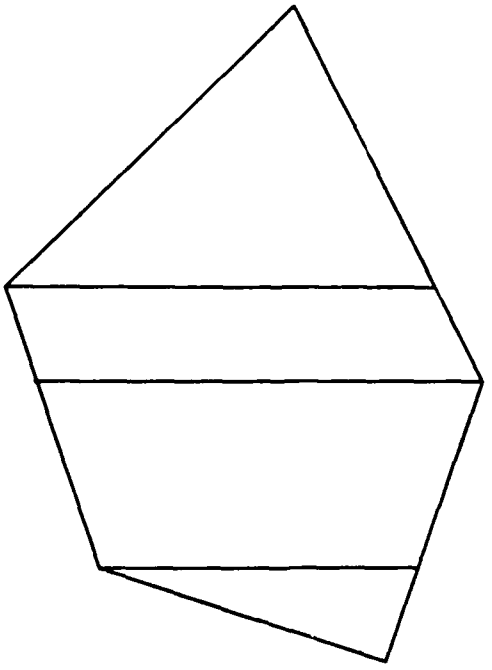


Figure 3.2. Polygon Decomposed into Four Trapezoids

least number of computations. For a trapezoid with four vertices, eleven pieces of information are required. All four vertices require x and z screen space values, and two vertices require y screen space values. Only two y screen space values are required because a trapezoid's top and bottom extents are both horizontal. Each trapezoid also requires a color value.

2. Gouraud shading. Gouraud shading requires twenty-two pieces of information. Each of four trapezoid vertices requires x and z screen space values, and two vertices require y screen space values. In addition, all four vertices require an RGB triple (there are three pieces of information for each RGB triple, one for each color component).

3. Phong shading. Phong shading requires thirty-four pieces of information. In addition to the information required for Gouraud shading, Phong shading requires twelve pieces of information. A three space normal, containing x, y, and z values in world space, at each of the four trapezoid vertices must be maintained.

Transforming object coordinates from one coordinate system to another (for example, world to eye coordinate conversion), and polygon rotations, scaling, and translations are also likely to benefit from vectorization.

There are several different ways to vectorize the z-buffer algorithm, and it is difficult to forecast which is likely to be the most efficient, and it is more likely that each method's results vary depending on the objects rendered. Four of the various options are:

- **Method 1: Edge & Scan-segment Interpolation with height and length buckets**

  1. Vector edge interpolation with height buckets
  2. Vector scan-segment interpolation with length buckets

- **Method 2: Edge & Scan-segment Interpolation with height buckets**

  1. Vector edge interpolation with height buckets
  2. Vector scan-segment interpolation without length buckets

- **Method 3: Edge & Scan-segment Interpolation with length buckets**

  1. Vector edge interpolation without height buckets
  2. Vector scan-segment interpolation with length buckets

- **Method 4: Edge and Scan-segment interpolation without buckets**

  1. Vector edge interpolation without height buckets
  2. Vector scan-segment interpolation without length buckets

The following describes the four methods in detail. Note that steps annotated with a "*" are implemented in vector form, and all of the below steps can be implemented in parallel.

3.4.2.1 Method 1: Edge & Scan-segment Interpolation with height and length buckets

1. Read the object and polygon descriptions.
2. Transform polygon local coordinates to world coordinates.
3. For all frames to be generated do #4 - #9
4. * Transform the resulting input polygons to the proper view by rotating, scaling, or translating, and autoscale.
5. * Transform all polygon world coordinates to eye coordinates.
6. For each polygon do:
   (a) Clip the polygon to the screen.
   (b) Purge if a backfacing polygon, perform flat shading calculations and perform perspective division.
   (c) Decompose the polygon into trapezoids. Care must be exercised when decomposing polygons which have colinear edges. Trapezoidal decomposition ensures that each edge is matched by only one other edge and reduces the requirement

for maintaining an active edge list and the associated sorting. Michael Goldapp states that "decomposition into trapezoids allows vector operations both horizontally and vertically," and this decomposition method is "nearly always best (Goldapp, 1986:141, 147)." The Silicon Graphics GTX has hardware support for this decomposition. In an article written about the GTX, the authors observed that "it will almost always be desirable to reduce the full polygons to collections of screen-aligned trapezoids whose parallel edges are in the direction of preferred fill in the frame buffer. (Akeley, 1988:241)." Silicon Graphics decomposes polygons into trapezoids with two parallel vertical edges, while Method 1 decomposes the polygons into trapezoids with two horizontal edges. Nothing favors one approach over the other (the required calculations are the same) with the exception that contiguous pixels in a scan-segment occupy contiguous elements in memory. Of course, contiguous elements in memory correspond to a Cray-2 memory stride of one. Note that the trapezoid decomposition proceeds in screen space, and each time the polygons are transformed for viewing the trapezoids must be recomputed.

(d) For each trapezoid do:

i. * Determine the height of each trapezoid (corresponding to the number of scan-lines which intersect a given trapezoid) and place each trapezoid into the height bucket. There is one height bucket for each possible trapezoid height value. Since there are 1024 lines on the Ultra display, the possible height values range from zero to 1023, so there are 1024 height buckets. Note that these buckets consume a large amount of memory. Each trapezoid consumes at most 34 words of memory (34 words for Phong shading, 22 words for Gouraud shading, 11 words for Flat shading), and each height bucket contains space for at most 64 trapezoids, so each height bucket requires 2176 words of memory. All of the height buckets together require 2,228,224 words of memory, which is equivalent to 17,825,792 bytes of Cray-2 memory. The vector length of the Cray-2 is 64, so it is more advantageous to operate on vectors with multiples of length 64, thus each height bucket

has space for 64 trapezoids.

ii. If the current bucket contains 64 trapezoids, empty the height bucket:

A. * Determine the right and left edge position values for all the trapezoids in the height bucket. (Edge values for color are also required if Gouraud shading is required; color plus surface normal are needed for Phong shading). Now each trapezoid in the bucket has been decomposed into scan-segments.

B. Place the resulting scan-segments into the appropriate length buckets. As with the height buckets, the length buckets consume a tremendous amount of memory. Each scan-segment consumes at most 16 words of memory (16 words for Phong shading, 10 words for Gouraud shading, and 5 words for flat shading). Notice that the memory requirement is smaller than for the trapezoid buckets. Each scan-segment bucket contains space for at most 64 scan-segments, and there must be one scan-segment bucket for each possible scan-segment length. Since there are 1280 pixels of horizontal resolution on the Ultra display, the possible length values range from zero to 1279. All of the scan-segment buckets consume 1,310,720 words of memory if Phong shading is used, or 10,485,760 bytes. By now, it should be evident that most computers do not contain enough main memory for such buckets. Even the Cray-1, with 2MW of memory would not contain enough memory for the height and length buckets. For computers which are limited in memory, dynamic memory allocation and disposal may lessen memory requirements somewhat. However, dynamic memory allocation may negatively impact vectorization due to the extensive use of pointers and lack of contiguous memory storage and the problem of unpredictable stride. Fortunately, the Cray-2 supercomputer provides the luxury of generous memory usage.

C. If the current scan-segment bucket contains 64 scan-segments, then perform z-buffering for all the scan-segments in the bucket by: 1) *

Calculating the z-depth and shading the scan-segment. For Gouraud shading, color must be interpolated across the scan-segment prior to shading calculations, while for Phong shading, surface normals must be interpolated across the scan-segment prior to shading. 2) Writing the scan-segment values to the refresh/frame-buffer.

7. Empty all height buckets which are not empty.
8. Empty all length buckets which are not empty.
9. Display the contents of the frame-buffer on the Ultra display.

3.4.2.2 Method 2: Edge & Scan-segment Interpolation with height buckets

1. Read the object and polygon descriptions.
2. Transform polygon local coordinates to world coordinates.
3. For all frames to be generated do #4 - #8
4. * Transform the resulting input polygons to the proper view by rotating, scaling, or translating, and autoscale.
5. * Transform all polygon world coordinates to eye coordinates.
6. For each polygon do:
   (a) Clip the polygon to the screen.
   (b) Purge if a backfacing polygon, perform flat shading calculations and perform perspective division.
   (c) Decompose the polygon into trapezoids.
   (d) For each trapezoid do:
      i. * Determine the height of each trapezoid (corresponding to the number of scan-lines which intersect a given trapezoid) and place each trapezoid into the height bucket.
      ii. If the current height bucket contains 64 trapezoids, empty the height bucket:

A. Determine the right and left edge position values for all trapezoids in the height bucket. This decomposes each trapezoid into a number of scan-segments. The numerical value of height is the number of scan-segments that must be interpolated for that trapezoid.

B. For all of the 64 scan-segments perform z-buffering: 1) * Interpolate each scan-segment in vector mode, producing pixel depth and color values for the scan-segment. 2) For all pixels contained in the scan-segment, if the current pixel depth is less than the z-buffer depth, write the pixel color value to the refresh buffer.

7. Empty all height buckets which are not empty.
8. Display the contents of the frame-buffer on the Ultra display.

3.4.2.3 Method 3: Edge & Scan-segment Interpolation with length buckets

1. Read the object and polygon descriptions.
2. Transform polygon local coordinates to world coordinates.
3. For all frames to be generated do #4 - #7
4. * Transform the resulting input polygons to the proper view by rotating, scaling, or translating, and autoscale.
5. * Transform the polygon world coordinates to eye coordinates.
6. For each polygon do:
   (a) Clip the polygon to the screen.
   (b) Purge if a backfacing polygon, perform flat shading calculations and perform perspective division.
   (c) Decompose the polygon into trapezoids.
   (d) For each trapezoid do:

i. * Individually interpolate each edge in vector, producing scan-segment values for the right and left trapezoid edges.

ii. Place each resulting scan-segment into the appropriate length buckets.

A. If the current scan-segment bucket contains 64 scan-segments, then for all the scan-segments in the bucket perform z-buffering: 1) * Calculate the z-depth and shade the scan-segment. 2) Write the scan-segment values to the refresh/frame-buffer.

7. Empty all length buckets which are not empty.

8. Display the contents of the frame-buffer on the Ultra display.

3.4.2.4 Method 4: Edge & Scan-segment Interpolation without buckets

1. Read the object and polygon descriptions.

2. Transform polygon local coordinates to world coordinates.

3. For all frames to be generated do #4 - #7

4. * Transform the resulting input polygons to the proper view by rotating, scaling, or translating, and autoscale.

5. * Transform the polygon world coordinates to eye coordinates.

6. For each polygon do:

(a) Clip the polygon to the screen.

(b) Purge if a backfacing polygon, perform flat shading calculations and perform perspective division.

(c) Decompose the polygon into trapezoids.

(d) For each trapezoid do:

i. * Individually interpolate each edge in vector, producing scan-segment values for the right and left trapezoid edges.

ii. * Interpolate individually each scan-segment in vector mode, producing pixel depth and color values for the scan-segment.

iii. For all pixels contained in the scan-segment, if the current pixel depth is less than the z-buffer depth value, write the resulting pixel color values to the refresh buffer.

7. Display the contents of the frame-buffer on the Ultra display.

3.5 Edge and Scan-segment Interpolation Methods

Edge and scan-segment interpolation are fundamental operations in this algorithm and they warrant further discussion which more rigorously defines the mathematical basis for the Pascal implementation. Linear interpolation is the process of determining the location of a point on a line given the two end points, and for edge and scan-segment interpolation, the desired result is a sequence of equally distant points along the edge or scan-segment. The polygon or trapezoid edges are generated by linear interpolation of planar polygons edges defined in screen space, and the scan-segments are generated from the end points of these edges, thus edge and scan-segment generation are linear interpolations.

In general, any sequence of equally distant points along a line segment defined by the end points (i_{start}, d_{start}) and (i_{end}, d_{end}) can be determined by the Equation 3.1:

$$d_{new} \leftarrow d_{start} + (d_{end} - d_{start}) \frac{i_{inter} - i_{start}}{i_{end} - i_{start}} \tag{3.1}$$

Where d_{new} is the value of the dependent variable at i_{inter}; and interpolation proceeds from i_{start} to i_{end}. Three different methods used for edge and scan-segment interpolation, based upon Equation 3.1, are listed below.

1. Scalar interpolation.
2. Bucketized interpolation.
3. Non-bucketized interpolation.

In this chapter, and in Chapter 4, Pascal vector syntax is used to express some of the thesis concepts and results. If Pascal syntax is unfamiliar, refer to the short overview provided in Appendix A.

3.5.1 Scalar interpolation The scalar method interpolates a line segment, defined by the end points (i_{start}, d_{start}) and (i_{end}, d_{end}), by repeated addition, using the recurrence relation given in Equation 3.2.

$$\begin{aligned} \Delta d &\leftarrow \frac{d_{end} - d_{start}}{i_{end} - i_{start}} \\ d_0 &\leftarrow d_{start} \\ d_n &\leftarrow d_{n-1} + \Delta d \qquad for \quad 1 \leq n \leq (i_{end} - i_{start}) \end{aligned} \tag{3.2}$$

In Equation 3.2, d is the dependent variable, i is the independent variable, and the distance $i_{end} - i_{start}$ is divided into $i_{end} - i_{start}$ intervals of equal length. Now, applying Equation 3.2 to edge interpolation, each edge is defined by the triples $(y_{top}, x_{top}, z_{top})$ and $(y_{bottom}, x_{bottom}, z_{bottom})$, where x is the horizontal position in three-space, y is the vertical position in three-space, and z is the depth in three-space. The x and y values are rounded to the nearest integers, since they correspond to pixels on the display. For edge interpolation, the independent variable i is the vertical position y, and the dependent variable d is either x or z. Thus:

$$\begin{aligned} i_{end} &\leftarrow y_{top} \\ i_{start} &\leftarrow y_{bottom} \\ d_{end} &\leftarrow (x_{top}, z_{top}) \\ d_{start} &\leftarrow (x_{bottom}, z_{bottom}) \end{aligned}$$

For edge interpolation of the x values, Equation 3.2 expressed in Pascal syntax is:

```
height := ytop - ybot;          { Polygon height (y extent of the edge)}
deltax := (xtop - xbot)/height;
x := xbot;
for i := 1 to height do         { Compute the x value where each      }
   x := x + deltax;             { scan-line intersects the edge.      }
```

Each scan-segment end point is defined by $(y_{left}, x_{left}, z_{left})$ and $(y_{right}, x_{right}, z_{right})$. The x and y values are integers. For scan-segment interpolation, the independent variable i is the horizontal position x, the dependent variable d is z, and y does not change across a scan-segment. Thus:

$$\begin{aligned} i_{end} &\leftarrow x_{right} \\ i_{start} &\leftarrow x_{left} \\ d_{end} &\leftarrow z_{right} \\ d_{start} &\leftarrow z_{left} \end{aligned}$$

Note that for Equation 3.2, Δd cannot be calculated for line segments of zero length ($i_{end} - i_{start} = 0$), but segments of zero length do not require interpolation. Scan-segments of zero length occur in graphics work, and usually correspond to one pixel or less of display coverage.

3.5.2 Bucketized interpolation When all trapezoid edges have the same non-zero height, or when all scan-segments have the same non-zero length, an entire bucket of size m can be interpolated by repeated addition using Equation 3.3, which is really a form of Equation 3.2.

$$\begin{aligned} \Delta d[1..m] &\leftarrow \frac{d_{end}[1..m] - d_{start}[1..m]}{i_{end}[1] - i_{start}[1]} \\ d_0[1..m] &\leftarrow d_{start}[1..m] \\ d_n[1..m] &\leftarrow d_{n-1}[1..m] + \Delta d[1..m] \qquad for \quad 1 \leq n \leq (i_{end}[1] - i_{start}[1]) \end{aligned} \tag{3.3}$$

For edge interpolation, the independent variable i is the vertical position y, and the dependent variable d is either x or z, and The height of the trapezoid is equivalent to $i_{end}[1] - i_{start}[1]$.

$$\begin{aligned} i_{end}[1..m] &\leftarrow y_{top}[1..m] \\ i_{start}[1..m] &\leftarrow y_{bottom}[1..m] \\ d_{end}[1..m] &\leftarrow (x_{top}[1..m], z_{top}[1..m]) \\ d_{start}[1..m] &\leftarrow (x_{bottom}[1..m], z_{bottom}[1..m]) \end{aligned}$$

For scan-segment interpolation, the independent variable i is the horizontal position x, the dependent variable d is the depthh z, the length of the scan-segment is equivalent to $i_{end}[1] - i_{start}[1]$, and the number of scan-segments in the bucket is m. Thus:

$$\begin{aligned} i_{end}[1..m] &\leftarrow x_{right}[1..m] \\ i_{start}[1..m] &\leftarrow x_{left}[1..m] \\ d_{end}[1..m] &\leftarrow z_{right}[1..m] \\ d_{start}[1..m] &\leftarrow z_{left}[1..m] \end{aligned}$$

For edge interpolation of the z values, the mathematical formula in Equation 3.3 can be expressed in Pascal syntax as:

```
deltaz[1..m] := (ztop[1..m] - zbot[1..m])/height;
z[1..m] := zbot[1..m];
for i := 1 to height do
   z[1..m] := z[1..m] + deltaz[1..m];
```

The Cray-2 can handle buckets of an arbitrary length, but optimum bucket size is a multiple of 64.

3.5.3 Non-bucketized interpolation Non-bucketized interpolation of edges and scan-segments is necessary when edges or scan-segments are not collected in buckets containing edges of identical height, or scan-segments of identical length, and when vector interpolation is desired. Equation 3.4 is a modification of Equation 3.2 such that there is no longer a recurrence relation. Eliminating the recurrence enables the operation to proceed in vector.

$$\begin{aligned} \Delta d &\leftarrow \frac{d_{end} - d_{start}}{i_{end} - i_{start}} \\ d_0 &\leftarrow d_{start} \\ \begin{bmatrix} d_0 \\ \vdots \\ d_n \end{bmatrix} &\leftarrow d_0 + \Delta d \times \begin{bmatrix} 0 \\ \vdots \\ n \end{bmatrix} \qquad for \quad n = i_{end} - i_{start} \end{aligned} \tag{3.4}$$

For edge interpolation, the independent variable i is the vertical position y, the dependent variable d is either x or z, and the height of the trapezoid is equivalent to $i_{end} - i_{start}$. Thus:

$$\begin{aligned} i_{end} &\leftarrow y_{top} \\ i_{start} &\leftarrow y_{bottom} \\ d_{end} &\leftarrow (x_{top}, z_{top}) \\ d_{start} &\leftarrow (x_{bottom}, z_{bottom}) \end{aligned}$$

For scan-segment interpolation, the independent variable i is the horizontal position x, the dependent variable d is computed for z, and the length of the scan-segment is equivalent to $i_{end} - i_{start}$. Thus:

$$\begin{aligned} i_{end} &\leftarrow x_{right} \\ i_{start} &\leftarrow x_{left} \\ d_{end} &\leftarrow z_{right} \\ d_{start} &\leftarrow z_{left} \end{aligned}$$

For scan-segment interpolation of the z values, the mathematical formula in Equation 3.4 is expressed in Pascal vector syntax as:

```
n := xright - xleft;                    { Scan-segment length          }
deltaz := (zright - zleft)/n;           { Reciprocal of the edge slope }
z[0..n] := zleft + deltaz * offset[0..n];
```

`Offset[0..n]` is an array filled with the values 0 through 1279. No offset greater than 1279 would be needed, since there are only 1280 pixels per horizontal raster display line.

Caution should be observed when vectorizing edge or scan-segment interpolation. Due to the limited floating point precision of any computer, the non-bucketized interpolation method may produce slightly different results than the scalar method. This was verified when non-bucketized interpolation was used in the vectorized software. Although

the image displayed on the screen showed no discernible differences between the two methods, an analysis of edge and scan-segment histograms produced using the two methods revealed some inconsistencies. (An edge histogram shows the number of trapezoids composing the object for any particular height, while a scan-segment histogram displays the number of scan-segments of any particular length. One would expect the histograms produced using the scalar method to match those produced using the vector method, but this was not the case. The histogram variations were small, but nonetheless evident).

3.6 Data Structures

Generally, the most time-consuming portion of a z-buffer is the edge and scan-segment interpolation. Because this interpolation is performed on trapezoids, the structure of the trapezoid data structure (Figure 3.3) is very important. According to Dyer and Whitman, who implemented a vectorized z-buffer on a Convex computer using the "C" programming language, data structures which are used to store data resulting from vector operations should minimize the use of pointers (Dyer and Whitman, 1987:38). Since the use of pointers makes the stride unpredictable, Pascal arrays or record structures should be used.

The trapezoid bucket data structure (Figure 3.4) closely resembles the trapezoid data structure so all of the trapezoid information can be placed into the bucket by a single Pascal statement. The record components are named differently to minimize ambiguity. The scan-segment bucket structure is depicted in Figure 3.5.

When using the Gouraud or Phong shading methods, the data structures in Figure 3.3, Figure 3.4, and Figure 3.5 require expansion. For Gouraud shading, color information for all four trapezoid vertices must be maintained, while for Phong shading, color and normal information for all four trapezoid vertices must be maintained.

```
type
  color    = 0..4294967295;

  trapezoidtype = record
    case t :boolean of
    true   :(ulex   : real;    {top left x value}
             uley   : integer; {top y value}
             ulez,             {top left z value}
             llex   : real;    {bottom left x value}
             lley   : integer; {bottom y value}
             llez,             {bottom left z value}
             urex,             {top right x value}
             urez,             {top right z value}
             lrex,             {bottom right x value}
             lrez   : real;    {bottom right z value}
             polcol : color);  {color}
    false  :(ta:array[1..11] of integer);
  end {record};

  traparr = array[0..8] of trapezoidtype;

var
  trapezoid : traparr;
```

Figure 3.3. Trapezoid Data Structure

```
type
  color    = 0..4294967295;
  mat64int = array[1..64] of integer;
  mat64rea = array[1..64] of real;
  mat64col = array[1..64] of color;

  heightbucket = record
    case t : boolean  of
    true   :(xltop    : mat64rea;  {top left x value}
             ytop     : mat64int;  {top y value}
             zltop,                {top left z value}
             xlbot    : mat64rea;  {bottom left x value}
             ybot     : mat64int;  {bottom y value}
             zlbot,                {bottom left z value}
             xrtop,                {top right x value}
             zrtop,                {top right z value}
             xrbot,                {bottom right x value}
             zrbot    : mat64rea;  {bottom right z value}
             polcol   : mat64col); {trapezoid color}
    false  : (tb:array[1..11,1..64] of integer);
  end {record};

  heightarr = array[0..1023] of heightbucket;
  heightcnt = array[0..1023] of integer;

var
  hgtbucket : heightarr;
  hgtcnt    : heightcnt;
```

Figure 3.4. Trapezoid Height Bucket Data Structure

```
type
  color    = 0..4294967295;
  mat64int = array[1..64] of integer;
  mat64rea = array[1..64] of real;
  mat64col = array[1..64] of color;

  lengthbucket = record
     lin,                      {active scanline}
     lpix        : mat64int;   {left pixel value}
     depl, depr  : mat64rea;   {left and right pixel depths}
     polcol      : mat64col;   {color of the scan-segment}
  end {record};

  lengtharr = array[0..1279] of lengthbucket;
  lengthcnt = array[0..1279] of integer;

var
  lgtbucket : lengtharr;
  lgtcnt    : lengthcnt;
```

Figure 3.5. Scan-segment Length Bucket Data Structure

3.7 Methodology

There are three different ways to vectorize the above design language:

1. Make use of the vectorizing compiler and write code in scalar form.
2. Write assembly code targeting the vector units.
3. Use the Pascal vector syntax.

With the first, one can write the code and allow the Cray-2 vectorizing compiler to vectorize the code. The Cray-2 Pascal vectorizing compiler allows switches to be set which automatically optimize scalar code, and attempts to vectorize code contained inside loop structures. Procedure calls defeat automatic vectorization. "If" statements do not necessarily defeat vectorization. An "if" statement can be vectorized provided the conditional expression is a vector expression and provided the "then" and "else" clauses contain only vector assignments (Conley, Oct 1988).

The one advantage of method one is that an extensive understanding of the Cray-2 architecture or Pascal vectorization syntax is not required. Unlike the Cray-2 FORTRAN compiler, however, the Pascal compiler does not indicate when a particular segment of code has been vectorized. Thus, the Pascal programmer cannot be certain that a particular segment of code has been vectorized unless the object code is examined.

The second method is to write assembly code for explicit vectorization by specifying in Cray Assembly Language (CAL) the loading of the vector registers and issuing of vector instructions. This method can produce code which executes faster than the corresponding code written in Pascal vector syntax, but its disadvantages include readability and maintainability problems that assembly language programming usually causes. Furthermore, it would be difficult to determine if the speedups were attributed to writing in assembly code or to using vectorization.

A reasonable alternative to methods one and two is to use the Pascal vector syntax to explicitly indicate where vectorization is to occur. Thus, the programmer is neither dependent on the automatic vectorization of scalar loop structures, which may or may not occur, nor is the programmer required to learn assembly code with its associated disadvantages. The z-buffer code was vectorized using this third method.

3.8 Summary

The Analysis and Design lays the groundwork for a systematic implementation and analysis of results. Since there are four implementation alternatives, each may be investigated, and the most efficient alternative chosen.

IV. Implementation and Results

4.1 Introduction

This chapter presents the thesis implementation and results in three sections. The first section, timing studies, explains the timing studies performed on the scalar and vectorized versions of the z-buffer code. Heuristics for vectorization, the second section, focuses on rules of thumb for vectorizing scalar code. These heuristics were discovered while vectorizing the z-buffer algorithm, but are applicable when vectorizing any algorithm. From experience gained by vectorizing the z-buffer code, the third section presents a different approach which should improve performance. This chapter concludes with a brief summary of the results.

4.2 Timing Studies

4.2.1 Introduction Timing studies are essential when comparing the relative performance of two or more different software versions. Timings can yield valuable information and reveal where improvements can be made. However, there are many things to consider when timing software, and poorly performed timing is not only useless, but may cause the algorithm designer to draw incorrect conclusions about algorithm performance. The following sections present the value of timing studies and how they are used in this thesis. They also discuss the machine dependency of timing studies, the statistical variation encountered when timing, and how timing is performed. Finally, display times for two different objects are given.

4.2.2 Timings are Hardware Dependent Timings are machine dependent. Consequently, when comparing the performance of a vectorized algorithm to its scalar counterpart, it is difficult to make generalizations which apply to a broad class of computer architectures. Dyer and Whitman implemented their z-buffer algorithm on a Convex C-1 supercomputer, using the Convex "C" vectorizing compiler, and they report a three to one speedup of the vectorized version over the scalar version (Dyer and Whitman, 1987:43). Depending on the relative efficiency of the scalar instructions to the vector instructions, such speedup may not be difficult to achieve.

4.2.2.1 Scalar Optimization When comparing the execution efficiency of a sequence of scalar instructions to their vector equivalent, it is necessary to understand how execution of the scalar instructions differs from vector instructions. The Cray-2 has several mechanisms for improving the efficiency of scalar instruction execution. These include a scalar optimizer and an architecture which permits several instructions to be executed simultaneously within the same processor. The Cray-2 computer contains nine independent functional units (adders, multipliers etc.), and unlike many computers, it can issue instructions when the required "functional unit, source and operand registers, and read/write data paths needed by the instruction are free at the particular clock periods that the instruction needs them. A resource does not necessarily need to be free at the clock period of issue because the execution of the instruction may not need the resource until a later clock period. Issue will proceed as long as the resource is available at the needed clock period (Conley, Nov 1988). If the instructions are properly ordered to eliminate dependencies (one instruction cannot depend on the results obtained in another simultaneously executed instruction), instructions can be executed in several different functional units at once, increasing performance significantly. Depending on the order and types of instructions, the scalar version of a program can execute faster if this parallelism is exploited. Thus, what the designer may consider to be relatively minor changes to the software, like reordering instructions, can have a profound impact on the performance of the scalar code.

The Cray Pascal scalar optimizer attempts to reorder instructions, eliminate unnecessary instructions, expand procedures inline, eliminate loop invariants and register transfers, and reduce memory use (Cray Pascal, 1986:13-12). At times, its precise behavior is unpredictable, and minor changes to the scalar code may have a significant effect upon performance. This can frustrate attempts to compare the relative efficiency of one scalar version to another, or the efficiency of the scalar version to the vector version. A pertinent question to ask is: if speedup occurs, is the speedup due to vectorization, or is it attributable to a reordering of the scalar instructions which triggers the scalar optimizer? Indeed, the designer is at the mercy of how the compiler writer chose to implement the programming language features, which by no means guarantees the implementation is an efficient one.

4.2.2.2 Programming Language Used to Vectorize Another important consideration is the programming language used to vectorize the algorithm. Performance comparisons of vector to scalar code across programming languages is complicated by compiler differences. Studies performed by the Air Force Weapons Laboratory show the performance of the relatively mature FORTRAN compiler to be significantly better than the performance of the Pascal compiler at both automatically vectorizing code and optimizing scalar code. The Pascal compiler also seems to generate "seemingly wasteful instruction sequences (Conley, Aug 1988)."

4.2.2.3 Software versus Hardware Algorithm Implementation As mentioned in Chapter 2, the z-buffer algorithm on the Cray-2 is implemented in software. Most workstations, however, have hardware z-buffer support. Algorithms imbedded in hardware can execute orders of magnitude faster than the same algorithms implemented in software. Comparing the efficiency of the Cray-2's architecture design to the that of the workstation's hardware design based on how quickly the Cray-2 can execute software instructions versus how quickly the workstations can execute instructions embedded in hardware is unreasonable.

4.2.3 Cray-2 Clock Functions The Cray-2 operating system supports a number of clock functions which return the execution time for a section of code. All of these timing functions exact a performance penalty, albeit small. Two different timing functions, the second function and the microsecond function, were used to time the code.

4.2.3.1 Second Function The second function uses CPU time to determine how long a given section of code takes to execute. The user is not affected when his job loses the processor. Accurate time, as measured by the second function, cannot be guaranteed for sections of code which take less than a millisecond.

4.2.3.2 Microsecond Function Unlike the second function, the microsecond function does not suffer from the inability to measure small time increments. It is based on the hardware real time clock, and is accurate to within one Cray-2 clock cycle, 4.1 ns. However, it continues to measure time when the job loses the processor (blocking), and is

very sensitive to system loading. Because the Cray-2 is heavily used, a job is not likely to have dedicated use of the processor and time measured by the microsecond function will vary according to the frequency with which the timed section of code is blocked. One way to circumvent the microsecond function's limitations is to have dedicated use of one of the Cray-2's processors, but this can be guaranteed only by prohibiting all others from using the computer – a rather costly solution. Therefore, the microsecond function should be reserved for those cases when the section of code under investigation takes less than several milliseconds, and the lowest time measured is the more accurate time. Averages of time should not be used, unless the user has dedicated use of the Cray-2.

4.2.4 Statistical Variation Random events can complicate timing studies. It is possible that all of the differences in timing between two pieces of code can be purely due to chance. To reduce this possibility, in most instances, five timing samples were observed and a mean taken for the five samples. In all cases, the deviation of the samples from the mean was small. Note that means determined from microsecond clock timings are likely to be inaccurate.

4.2.5 Timing Caveats Clearly, timings are data dependent. Conclusions drawn from one set of data may conflict with those drawn from another, so it is usually desirable to perform timings on different data sets. Analysis of different data sets should suggest limitations of the vector software and may indicate which type of data is best operated upon with either vector or scalar software. With this is mind, two different data sets were used. One of the data sets is an F-16 aircraft, characterized by small polygons in the normal view (the view used for testing). The other data set describes a dodecahedron (a polyhederon with twelve faces) and an icosahederon (a polyhederon with twenty faces) combination, characterized by large polygons. Of course, polygon size is view dependent, and either the F-16 (Figure 4.2.5) or the "DodecaIcosahederon" (Figure 4.2.5) viewing parameters could be changed resulting in different polygon statistics.

The F-16 data was generated by Major Larry Feldman of the Air Force Weapons Laboratory, and the "DodecaIcosahederon" data was created by Mr. Bob Conley. Relevant statistics for the F-16 and the "DodecaIcosahederon," important because they indicate

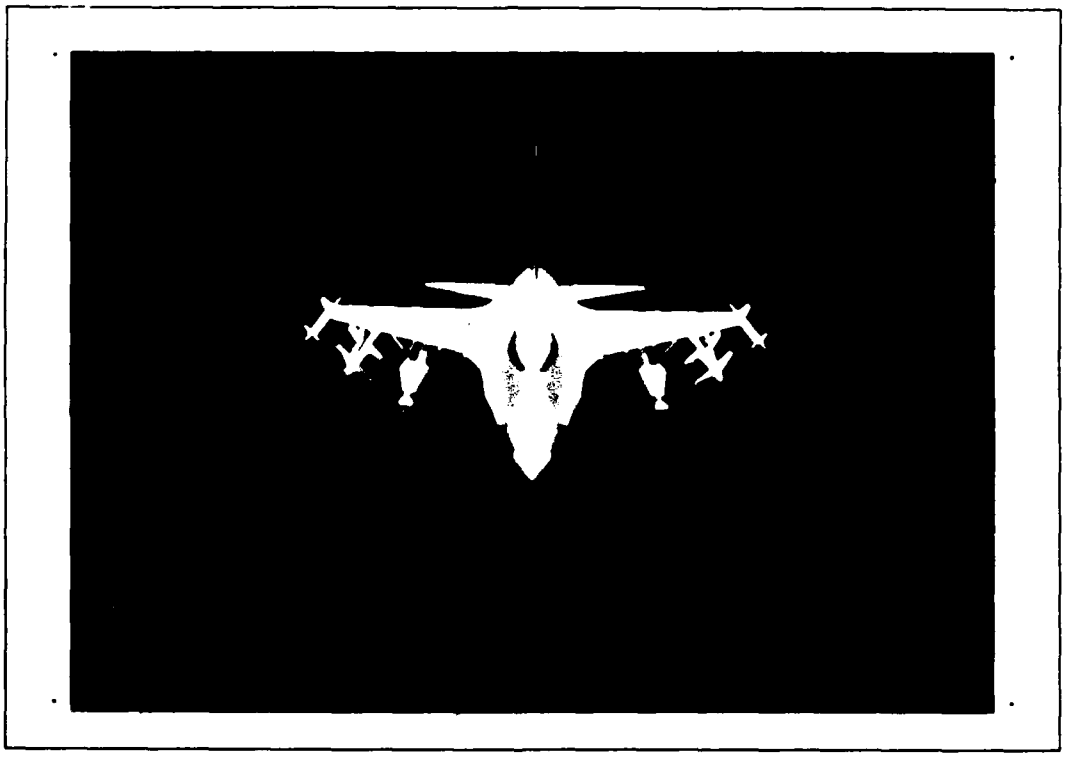


Figure 4.1. F-16 in the Normal View

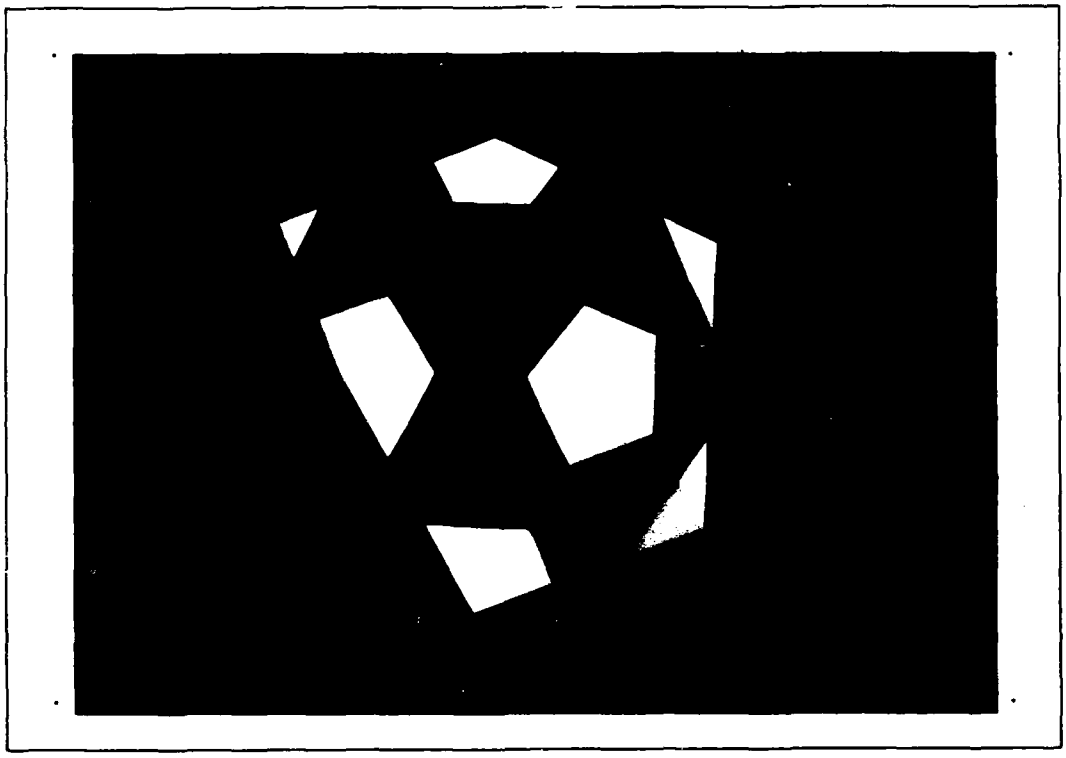


Figure 4.2. "DodecaIcosahederon in the Normal View

- Number of polygons 2997
- Number of pixels 193352
- Number of scan-segments 44588
- Number of scan-segments rejected 248
- Mean number of pixels per polygon 64.52
- Mean scan-segment length (pixels) 4.36
- Mean polygon height (pixels) 14.79

Table 4.1. F-16 Polygon Statistics

- Number of trapezoids 7644
- Number of pixels 219625
- Number of duplicate pixels 26273
- Number of scan-segments 49262
- Number of scan-segments rejected 382
- Mean number of pixels per trapezoid 28.73
- Mean scan-segment length (pixels) 4.49
- Mean trapezoid height (pixels) 6.39
- Mean number of trapezoids per polygon 2.55

Table 4.2. F-16 Trapezoid Statistics

vector lengths and aid in timing analyses, are shown in Tables 4.1 through 4.4.

Statistics reported for each object include the number of polygons and the number of trapezoids that compose the object. Polygons are decomposed into trapezoids, and due to the nature of the decomposition, some of the scan-segments are shared across trapezoid edge boundaries. The "number of duplicate pixels" statistic reports the extra number of pixels which must be computed when working with trapezoids. Another interesting statistic is the number of scan-segments rejected. These scan-segments are of a negative length (the right edge pixel value is less than the left edge pixel value). This anomaly is due to some of the polygons not being rejected as backfacing polygons because of limited floating point precision in the purge back-facing polygon procedure.

- Number of polygons 13
- Number of pixels 555505
- Number of scan-segments 4131
- Number of scan-segments rejected 347
- Mean number of pixels per polygon 42731.15
- Mean scan-segment length (pixels) 146.80
- Mean polygon height (pixels) 291.08

Table 4.3. "DodecaIcosahederon" Polygon Statistics

- Number of trapezoids 36
- Number of pixels 560944
- Number of duplicate pixels 5439
- Number of scan-segments 3807
- Number of scan-segments rejected 349
- Mean number of pixels per trapezoid 15581.78
- Mean scan-segment length (pixels) 147.35
- Mean trapezoid height (pixels) 105.75
- Mean number of trapezoids per polygon 2.77

Table 4.4. "DodecaIcosahederon" Trapezoid Statistics

4.2.6 How the Code was Decomposed for Timing For timing purposes, the vector and scalar versions of the z-buffer software are broken into six areas:

1. Rotation, scaling, and translation.
   (a) This section requires the multiplication of two four by four floating point matrices.
   (b) The microsecond clock was used to time this area because the time to execute the instructions was so small.
2. World to eye coordinate conversion.
   (a) This section requires multiplying all the three-dimensional points representing the polygons by a transformation matrix.
   (b) This was timed using the second clock.
3. Miscellaneous overhead.
   (a) This includes perspective division, lighting calculations, and the purging of back-facing polygons.
   (b) This was timed using the second clock.
4. Trapezoidal Decomposition.
   (a) Breaks trapezoids into polygons.
   (b) This was timed using the second clock.
5. Polygon/trapezoid edge interpolation. Polygon/trapezoid edge interpolation was timed using the second clock and can be performed three different ways. These are:
   (a) Scalar method.
   (b) Height bucket method.
   (c) Individual edge interpolation.

6. Scan-segment interpolation. Scan-segment interpolation was timed using the second clock and can be performed three different ways. These are:

   (a) Scalar method.

   (b) Scan-segment bucket method.

   (c) Individual scan-segment method.

Each area was separately timed because vectorization may cause speedup in some areas and slowdown in others. Speedup is measured as: $Time_{scalar}/Time_{vector}$ and speedup greater than one indicates that vectorization is more efficient. Generally, timings were collected by enclosing comments around all portions of the code except for the area of interest. Caution must be exercised here. For example, if the overhead for a procedure call is desired, the designer may be tempted to time the execution of the procedure and then time the execution with the inner body of the procedure commented out. This may produce misleading results, as the scalar optimizer may detect that the procedure contains no executable instructions, and eliminate the entire procedure call.

Another rather obvious requirement is for both the scalar and vector software to produce the same results. In computer graphics work, this is relatively easy to verify, since the images produced by each software version can be compared. In other work, some sort of verification must be used.

Upon analysis of the four vector methods presented in Chapter 3, it is evident that all four methods do not have to be individually coded and tested to determine which methods are more efficient. It is sufficient to break the significant areas of interest into code sections and to time the execution of each section. Then, the four vector methods are reconstructed by adding up the execution times for each of the sections. In this manner, the timing analysis performed on each section will give a rough idea of the most efficient method.

The four vector methods differ from each other only in the method of edge and scan-segment interpolation. An indication of the value of each method can be determined by timing edge interpolation with and without height buckets, and by timing scan-segment

interpolation with and without scan-segment buckets. The execution time for rotation, scaling, and translation, world to eye coordinate conversion, miscellaneous overhead, and trapezoidal decomposition for all four vector methods remains the same.

4.2.7 Timings for the F-16 This section presents the timing results for displaying the F-16. An asterisk "*" indicates that the code was vectorized.

4.2.7.1 Scalar Timing - no clipping

- Rotation, scaling, and translation ..... 861.87 μs
- World to eye coordinate conversion ..... 21.50 ms
- Total of above and overhead ..... 30.00 ms
- Miscellaneous display overhead ..... 201.00 ms
- Trapezoidal Decomposition ..... 0.00 ms
- Polygon edge interpolation ..... 248.00 ms
- Scan-segment interpolation ..... 341.00 ms
- Total display time ..... 790.00 ms
- Total ..... 820.00 ms

4.2.7.2 Vector Timing - no clipping

- *Rotation, scaling, and translation ..... 066.88 μs
- *World to eye coordinate conversion ..... 1.00 ms
- Total of above and overhead ..... 11.50 ms
- Miscellaneous display overhead ..... 204.00 ms
- Trapezoidal Decomposition ..... 145.00 ms
- *Trapezoid edge interpolation (buckets) ..... 132.00 ms
- *Trapezoid edge interpolation (no buckets) ..... 157.10 ms
- *Scan-segment interpolation (buckets) ..... 483.00 ms
- *Scan-segment interpolation (no buckets) ..... 507.80 ms
- **Method 1 total display time** ..... 964 ms
- **Method 2 total display time** ..... 989 ms
- **Method 3 total display time** ..... 989 ms
- **Method 4 total display time** ..... 1014 ms

Section	Scalar	Vector		Speedup
*Rotation etc.	861.87 μs	066.80 μs	(< 1%)	12.90
*World to eye conversion	21.50 ms	$\approx$ 1.00 ms	(< 1%)	21.50
Above and overhead	30.00 ms	11.50 ms	(1.2%)	2.61
Miscellaneous overhead	201.00 ms	204.00 ms	(21.2%)	0.99
Trapezoidal Decomposition	0.00 ms	145.00 ms	(15.0%)	0.00
*Edge interpolation	248.00 ms	132.00 ms	(13.7%)	1.89
*Scan-seg interpolation	341.00 ms	483.00 ms	(50.1%)	0.71
Total display time	790.00 ms	964.00 ms		0.82

Table 4.5. Speedup Table for F-16 Display

4.2.7.3 Summary of F-16 Timings Unpredictably, the scalar version of the program executed faster than any of the vectorized versions. This result is particularly perplexing, especially since vectorization did produce some localized speedups. The scalar version (without clipping) took 790 ms to display the F-16, while the fastest fully vectorized version, vector method one took 964 ms, for a speedup of 0.820. Because speedup was not achieved, it was necessary to determine where the vectorized code performed more poorly than the scalar code. Table 4.5 lists the performance and speedup of the scalar code over the vector code, and the percentage of time consumed by each section of the vectorized code.

Table 4.5 indicates that there are at least two areas which require further attention: trapezoidal decomposition and scan-segment interpolation. Trapezoidal decomposition is not required in the scalar version but is essential for interpolating polygon edges in vector. Notice that trapezoidal decomposition is not implemented in vector, and cannot be effectively vectorized unless all polygons consist of a certain number of edges. Trapezoidal decomposition could be vectorized if all polygons were triangles, requiring all input polygons to be decomposed into triangles (Conley, Jul 88). Converting the polygons to triangles could occur in world space once, and never again. Except for the generation of the first frame of an animation, polygon to triangle conversion would not be performed. This time is likely to be a small percentage of the total display time and should not appreciably affect the ability to achieve real-time performance. Once the triangles were generated, they could be decomposed into trapezoids in vector mode.

The other area which must be considered is the scan-segment interpolation, which consumes around 50% of the display time. At first, the cause of this performance degradation was believed to be the duplication of scan-segments along trapezoid edge borders. Scan-segment duplication occurs because the bottom scan-segment of one trapezoid is identical to the top scan-segment of its lower neighboring trapezoid. After analyzing the F-16 statistics provided in Tables 4.1 and 4.2, the number of duplicate pixels which must be interpolated is only 12% of the total. Since the amount of time to interpolate a scan-segment is proportional to the number of pixels covered, it is reasonable to expect only a 12% increase in the scan-segment interpolation time. Furthermore, the average polygon is only 64.52 pixels, and the ratio of duplicate scan-segments is less when the polygons are larger, since the number of duplicate scan-segments increases with the perimeter of the polygon, while the total number of pixels increases with the area of the polygon. Due to the small average size of the polygons, a 12% duplication of pixels is likely to be a large ratio. Consider the "DodecaIcosahederon," with an average polygon size of 42,731 pixels. The number of pixels occurring within duplicate scan-segments is only 1.2% of the total.

Eliminating the duplicate scan-segments requires modifying the trapezoidal decomposition routine, which would certainly result in longer execution time for this routine. Thus, the savings from not interpolating duplicate scan-segments must be greater than the additional time spent eliminating these segments. With such a low percentage of time spent interpolating duplicate scan-segments, a modification of the trapezoid generation code was deemed inefficient.

Unlike the scan-segment interpolation, edge interpolation benefited from vectorization. Bucketized edge interpolation took 132 ms, and achieved a speedup of almost two times over the scalar version. Non-bucketized edge interpolation, which took 157.10 ms, was faster than the scalar edge interpolation, but slower than the bucketized edge interpolation.

Most encouraging were the speedups for rotation, scaling, and translation and world-to-eye coordinate transformations: 12.90 and 21.50 respectively. Both of these areas involve extensive use of matrix algebra, which can be vectorized rather handily. However, both consume such a small portion of overall execution time, around 0.1%, that the speedup had

a negligible effect upon the whole program. It should be evident by now that slowdown in one area can mask speedup, reinforcing the importance of timing individual sections of code.

4.2.8 Timings for the "DodecaIcosahederon" This section presents the timing results for displaying the "DodecaIcosahederon."

4.2.8.1 Scalar Timing - no clipping

- Rotation, scaling, and translation ..... < 1.00 ms
- World to eye coordinate conversion ..... < 1.00 ms
- Total of above and overhead ..... < 1.00 ms
- Polygon edge interpolation ..... 11.00 ms
- Scan-segment interpolation ..... 726.40 ms
- Total display time ..... 738.40 ms
- Total ..... 739.40 ms

4.2.8.2 Vector Timing - no clipping

- *Rotation, scaling, and translation ..... < 1.00 ms
- *World to eye coordinate conversion ..... < 1.00 ms
- Total of above and overhead ..... < 1.00 ms
- Trapezoidal decomposition ..... < 1.00 ms
- *Trapezoid edge interpolation (buckets) ..... 25.47 ms
- *Trapezoid edge interpolation (no buckets) ..... 11.10 ms
- *Scan-segment interpolation (buckets) ..... 1158.60 ms
- *Scan-segment interpolation (no buckets) ..... 1029.80 ms
- Method 1 total display time ..... 1185 ms
- Method 2 total display time ..... 1056 ms
- Method 3 total display time ..... 1171 ms
- Method 4 total display time ..... 1042 ms

Section	Scalar	Vector		Speedup
*Rotation etc.	< 1.00 ms	< 1.00 ms	(< 1%)	
*World to eye conversion	< 1.00 ms	< 1.00 ms	(< 1%)	
Above and overhead	< 1.00 ms	< 1.00 ms	(< 1%)	
Miscellaneous overhead	< 1.00 ms	< 1.00 ms	(< 1%)	
Trapezoidal Decomposition	< 1.00 ms	< 1.00 ms	(< 1%)	
*Edge interpolation	11.00 ms	11.10 ms	(01.1%)	0.99
*Scan-seg interpolation	726.40 ms	1029.80 ms	(98.9%)	0.71
Total display time	738.40 ms	1041.90 ms		0.71

Table 4.6. Speedup Table for "DodecaIcosahederon"

4.2.9 Summary of "DodecaIcosahederon Timings' Similar to the F-16 timing results, the scalar version of the program executed faster than any of the full vectorized versions. There were no measurable localized speedups. Table 4.6 lists the performance and speedup of the scalar code over the vector method four code. Note that almost all of the time was consumed by scan-segment interpolation (almost 99%), and the time spent in trapezoidal decomposition is negligible. Since the number of polygons composing the "DodecaIcosahederon" is small, trapezoidal decomposition, as well as coordinate transformation, perspective division, and lighting calculations do not consume much time.

4.3 Heuristics for Vectorization

4.3.1 Introduction An heuristic is a rule of thumb, or an educated guess used for problem solving. This section investigates rules which generally result in the speedup of vectorized code over scalar code and also discusses the advantages and disadvantages of vectorization, and follows this with twenty heuristics for vectorization.

4.3.2 Advantages and Disadvantages of Vectorization Effective vectorization is still a black art, and requires some of the most difficult design and coding that I have experienced, even more difficult than parallelizing code for implementation on several processors. When vectorizing, not only is the computer scientist responsible for efficient algorithm design, but he must target this design to a specific architecture. In the absence of a superb automatic vectorizing compiler, vectorization without a good understanding of the target

architecture is likely to go awry. Vectorization is very much a trial and error process, and sometimes it defies intuition. It is generally advantageous to try several different vector alternatives, which was the intent of describing the four different vector methods presented in Chapter 3. Predicting which among several different vector alternatives is likely to be more efficient may not work.

Poor vectorizing compilers also quickly frustrate the diligent vectorizer, but compilers are like any other software product, and may be inefficiently designed. While automatic vectorizing compilers will improve, they do not compensate for poor algorithm design, and for best results, the computer scientist must target an algorithm for either scalar or vector operation. Nonetheless, despite the difficulties inherent in vectorization, it can provide tremendous speedups when applied to the appropriate problems and when done properly.

Like all methods for improving code efficiency, the advantages must be weighed against the disadvantages. There are several areas against which vectorization must be evaluated before a designer should determine to vectorize code. Most importantly, vectorized software must not transform well designed scalar code into an unintelligible mess. David Fisher lists symptoms of poor software. He states: "symptoms of the software crisis appear in the form of software that is non-responsive to user needs, unreliable, excessively expensive, untimely, difficult to maintain, and not reusable (Fisher, 1978:26)." It would be foolish to evaluate the merits of vectorization without directing attention at these areas. Five areas: reliability, cost, maintainability, portability, and efficiency are discussed below. Note that these five areas apply for code which is *explicitly vectorized*, and *not* for code which is written in scalar form and automatically vectorized by the compiler.

1. *Reliability.* Vectorized software can certainly be as reliable as scalar software, but because it is usually more difficult to design and code, reliable vectorized software is likely to be more difficult to achieve than reliable scalar software.
2. *Cost.* Generally, software developed to exploit a computer's architecture takes longer to develop than software which is developed for the general case. Since software cost is directly proportional to the time spent developing it, and vector software generally

takes longer to develop than its scalar equivalent, vector software will probably more expensive than scalar software.

3. *Maintainability and readability.* This area is where vectorized software can be a liability. Vector software is usually more difficult to read than scalar software, and what the software is doing may not be easily apparent from the code. It requires data structures which are designed for vectorization efficiency rather than data structures which are designed to be an abstraction of the problem. Less readable software is generally less maintainable software, since the maintenance programmer must understand the software before it can be modified. Vector software also contributes to the proliferation of programming language dialects, as vector syntax must be supported in the programming language when explicit vectorization is a desired feature. With vectorization a known feature for enhancing code efficiency, new programming languages should incorporate vector syntax as part of the language. For computers which do not have vector architectures, compilers could convert the vector syntax into a sequence of equivalent scalar instructions. Ada compilers do this when implementing slice instructions.

4. *Portability.* Vectorized software is unlikely to be portable from one computer architecture to another, since effective vectorization targets computer architecture features.

5. *Efficiency.* This is where vectorized software has its primary advantage over scalar software. Sometimes the computer scientist has no choice but to radically improve software to achieve particular timing goals, like real-time display. He can resort to scalar assembly code programming to achieve this end, or vectorization may be an option.

4.3.3 Twenty Heuristics for Producing Vectorized Software Some of the heuristics which are presented here are applicable only to the Pascal implementation on the Cray-2. Many of them, however, apply regardless of the machine for which the software is targeted.

1. *Implement in scalar first and vectorize only when necessary.* Implementing the scalar version is usually less labor intensive than implementing the equivalent vector ver-

sion. The scalar version is also more likely to be more maintainable and readable. Furthermore, the scalar version gives the designer an idea of how the problem can be solved, and if the scalar version does not meet performance requirements then it can be vectorized. Should vectorization become necessary, then the algorithm and data structures used in the scalar version may need to change. However, if the scalar version meets performance requirements, the disadvantages of vectorization dictate that the code not be vectorized. Yourdon notes that "since only a small portion of code is significant in terms of efficiency, the strategy should be to get the system working, then isolate the inefficient parts and optimize them (Yourdon,1979 : 102)."

2. *Try porting to a faster machine.* In the event that performance requirements are not met and if another machine is available with a faster processor, faster memory, or a better compiler, port the scalar code to that machine. The faster machine may provide the necessary speedup. Basically, the goal is to keep the code as simple and readable as possible. Investigate the possibility of parallelizing code, if a parallel solution is straight-forward and easier to implement than a vector solution.

3. *Make use of the vectorizing compiler.* Some vectorizing compilers do a superb job of automatically vectorizing "for loops," and if they are used, a minimal knowledge of the computer architecture is necessary. (Preliminary studies performed by Bob Conley at the Air Force Weapons Lab indicate that the Cray-2 FORTRAN compiler is better at automatically vectorizing "for loops" than using FORTRAN explicit array syntax, which is not the case for the Cray-2 Pascal compiler.) Write the code in scalar form, and allow the compiler to vectorize the "for loops." This does not eliminate the requirement to write "for loops" in a form that permit vectorization. Remember that procedure calls and "if" statements may defeat vectorization, and thus some modularity may have to be sacrificed for efficiency.

4. *Watch out for errors in compiler implementations.* Vectorizing compilers are difficult to write and may not correctly implement all features of the vector syntax as indicated in the reference manual. The Cray-2 Pascal compiler is a relatively immature compiler, and in the course of this thesis, several errors were discovered.

(a) One of the errors discovered involves the use of the array merge syntax in the scan-segment interpolation routine:

```
image^[cel[1..nscans]] :=
  if depl[1..nscans] < zbuff[cel[1..nscans]] then
    polcol[1..nscans]
  else image^[cel[1..nscans]];
```

This code causes a run-time error (memory interrupt) when executed. "`image^`" is a packed array containing pixel color values. This statement had to be replaced by:

```
for j := 1 to nscans do
   if depl[j] < zbuff[cel[j]] then
    image^[cel[j]] := polcol[j];
```

The compiler cannot vectorize packed array structures. However, when vector syntax is used with a packed array, the compiler should always disable vectorization and replace the vector implementation with the scalar equivalent.

(b) The compiler also seems to be tricked by nested record structures. Examine the record declared in Figure 4.3. Accessing "`vlist[1..numv].w.x`" produced incorrect values, but the semantic equivalent "`vlist[1..numv].a[3]`" produced the correct values.

5. *Design data structures carefully, paying attention to stride.* Design data structures with a stride of one between accessed elements. A vector memory access instruction attempts to grab, for instance, 64 elements from memory. By keeping the stride low, and avoiding strides that are powers of two, efficiency is enhanced.

(a) Compare the data structure in Figure 3.4, Trapezoid Height Bucket Data Structure, which has a stride of one between successive variable references, (for example, element one of "`xltop`" to element two of "`xltop`") to the data structure in Figure 4.4, which has a stride of 11 between successive elements. The data structure in Figure 3.4 is the more desirable.

```
const vmax   = 32767;       { Max num of ctl pts  }

type
  point3d =    record x,y,z : real;  end;
  vertex  =    record
    case t : boolean of
      true   : (
       cf    : boolean;     { Color flag         }
       nf    : boolean;     { Normal flag        }
       w     : point3d;     { World space triple }
       e     : point3d;     { Eye space triple   }
       col   : tristim;     { Color tristimulus  }
       nrm   : point3d);    { Surface normal     }
      false  : (
       a     : array[1..14] of real);
  end;

  {The a array in the false variant part overlays}
  { the elements declared in the true part       }

  varr = array [0..vmax] of vertex;

var
  vlist : varr;
  numv : integer;
```

Figure 4.3. Nested Record Data Structure

```
type
  color = 0..4294967295;

  heightbucket = record
    case t : boolean  of
    true   :(xltop    : real;      {top left x value}
             ytop     : integer;   {top y value}
             zltop,                {top left z value}
             xlbot    : real;      {bottom left x value}
             ybot     : integer;   {bottom y value}
             zlbot,                {bottom left z value}
             xrtop,                {top right x value}
             zrtop,                {top right z value}
             xrbot,                {bottom right x value}
             zrbot    : real;      {bottom right z value}
             polcol   : color);    {trapezoid color}
    false  : (tb:array[1..11] of integer);
  end {record};

  heightarr = array[0..1023] of heightbucket;
  heightcnt = array[0..1023] of integer;

var
  hgtbucket : heightarr;
  hgtcnt : heightcnt;
```

Figure 4.4. Alternate Trapezoid Height Bucket Data Structure

(b) In a similar case, two different trapezoid height bucket record structures were constructed, one with a stride of one and one with a stride of 35. The program using the data structure with a stride of one executed 7% faster than the other.

6. *Be aware of multidimensional array access order.* Different programming languages access array elements differently. Contiguous elements in memory are referenced by incrementing the leftmost array subscript in a FORTRAN array, while Pascal contiguous elements in memory are those referenced by successive indices in the rightmost part of the array. In other words, when accessing FORTRAN arrays, increment the leftmost array subscript most frequently, and when accessing Pascal arrays, increment the rightmost array subscript most frequently. For example, the following loop structure ensures a stride of one through the Pascal array:

```
type aarray = array[1..256,1..64] of real;

var
  a, b : aarray;
  i, j : integer;

  {equivalent to a[1..256,1..64,1] := b[1..256,1..64];}

  for i := 1 to 256 do
     for j := 1 to 64 do
        a[i,j] := b[i,j];
```

However, the following loop structure forces a stride of 256, the worst possible:

```
for i := 1 to 64 do
   for j := 1 to 256 do
      a[j,i] := b[j,i];
```

7. *Minimize memory access.* Retrieving large vectors from memory and storing large vectors can be time consuming. Try to perform all necessary operations on a particular vector all at once, and store it at the end. In other words, minimize the use of temporary variables. Given the following statements:

```
type aarray = array [1..256] of real;

var
  a, b, c : aarray;
  i, j    : integer;

  a := b * j;
  c := a * i;
```

If the optimizer does not catch the presence of the unnecessary store of "a," then the two above assignments should be replaced with "`c := b * j * i;`."

8. *Do not use call by value parameters.* Modern software engineering practices specify using call by value formal parameters when the actual parameter value is not to be changed, eliminating the possibility of unintentionally changing the actual parameter. Call by value parameters require a copy of the actual parameter to be made at the procedure call. For parameters which consume small amounts of memory, the overhead of making copies is negligible. However, for large vectors, this overhead is significant. For large vectors, use call by reference parameters. A copy of the vector is not made, but instead, a pointer to the vector is passed to the called procedure. In Pascal, call by value is the default parameter passing mode, and call by reference is specified by using "**var**" in the formal parameter list. Originally, the z-buffer program passed the world coordinate vector, a vector with 8000 elements, to the world-to-eye coordinate conversion procedure by value. It took 35.5 ms to perform the conversion. When this vector was passed using call by reference, the procedure took around 1 ms.

9. *Work with large arrays.* The Cray-2's vector registers can hold 64 64-bit vectors. When possible, save up elements until there are 64. This was the purpose of the height (edge) buckets and the length (scan-segment) buckets used in the z-buffer program. Edges or scan-segments of the same length were accumulated until there were 64. Then the buckets were emptied and the vector operations were performed. Generally, the larger the array, the better.

Note, however, that using buckets is not always advantageous. They improve performance only when the extra processing required to fill them is offset by the efficiency gains of having the Cray-2 operate on large vectors. In many cases, this extra processing to fill the buckets must proceed in scalar mode, and consequently, performance suffers. While the F-16 display time was faster for bucketized interpolation than for non-bucketized interpolation, the "DodecaIcosahederon" display time was faster when non-bucketized interpolation was used.

10. *Avoid the requirement for type conversions.* When the right side of an assignment statement contains a variable of a different type than the left side the statement, a type conversion must occur. Type conversions slow vector operations. Generally, this is not a problem, since Pascal does not permit incompatible type assignments. However, Pascal permits the assignment of an integer variable to a real variable. For example, suppose the following vector operation is performed:

```
pixdep[0..length] := depl + pixpos[0..length] * ddep;
```

and "**pixdep**," "**depl**," and "**ddep**" are real variables, while "**pixpos**" is an integer array. "**Pixpos**" must undergo a type conversion from integer to real. By varying "**length**" from 0 to 100 with "**pixpos**" declared an integer array, the statement executed 22% to 53% worse than if "**pixpos**" were declared a real array; and as "**length**" grew, the statement with the integer "**pixpos**" performance worsened.

11. *Time individual sections of code.* The rotation, scaling, translation and world-to-eye coordinate conversion sections of code consume such a small portion of the total processing time that their effect on the efficiency of the whole program is negligible, reinforcing the importance of timing individual sections of code to find where significant portions of time are spent and concentrating on vectorizing them. Indeed, the old axiom that 10% of the code takes 90% of the time seems to apply (Yourdon, 1979:102).

12. *Vectorization usually requires extra processing.* Most vectorization requires some overhead processing which is not required for the equivalent scalar implementation. The time taken for overhead processing must be offset by the speedup achieved by vectorization. For the vector z-buffer, the decomposition of polygons into trapezoids, not required for the scalar z-buffer, was pure overhead processing which enabled vectorized edge interpolation. Unfortunately, the speedup achieved did not compensate for the extra processing required.

13. *Understand the order of "if statement" evaluations.* Array merges cause problems when there are exceptional conditions.

    (a) For example, the following statement causes an execution error - "divide by zero" when any element of "**dpix**" is zero:

```
ddep[1..ntraps] :=
if dpix[1..ntraps] > 0 then
  depr[1..ntraps]-depl[1..ntraps]/dpix[1..ntraps]
else 0.0;
```

    The "`if depr[1..ntraps]-depl[1..ntraps]/dpix[1..ntraps]`" and "`else 0.0`" clauses are evaluated first; then the "`if dpix[1..ntraps] > 0`" is evaluated to determine which results to place in the left side of the expression.

    (b) The following is a method to treat exceptional conditions:

```
if All(k[1..length] <> 0.0) then
  r[1..length] := 1.0 / k[1..length]
else Halt;
```

    If any of the elements of "**k**" is 0.0, then the program halts, preventing division by zero. However, if special processing is required for any elements which are 0.0, instead of halting the program, then each element in the "**k**" array must be individually inspected in scalar form.

14. *Avoid data dependencies.* Expressions with data dependencies cannot be vectorized reliably. For instance, the expression:

```
hgtcnt[index[1..length]] := hgtcnt[index[1..length]] + 1;
```

does not give reliable results when two or more elements of "`index[1..length]`" are identical. If "`index`" contains the values "`1, 3, 4, and 1`", then we would expect "`hgtcnt[1]`" to be incremented by "2" and "`hgtcnt[3]`" and "`hgtcnt[4]`" to be incremented by "1". However, this does not occur when the value to be stored in "`hgtcnt[1]`" does not have time to complete the vector operation and be stored before the old value of "`hgtcnt[1]`" enters the pipeline.

15. *Enable range checking to minimize debugging.* The Pascal compiler has an array index range checking capability (the r+ option) which prevents array indices from exceeding their bounds. Range checking must be disabled for vectorization to occur. When developing code, enable range checking, otherwise array bounds may be exceeded without a run-time error. Once the program generates the correct results, disable range checking so the code can be vectorized.

    (a) For example, when range checking is disabled, the following potential run-time error, triggered when "`ntraps`" is greater than 64, may not be caught:

```
type mat64rea = array[1..64] of real;

var dxl, xltop, xlbot : mat64rea;

  dxl[1..ntraps] := xltop[1..ntraps] - xlbot[1..ntraps];
```

    (b) For an indication of how much quicker a section of code can run with range checking disabled (and vectorization enabled) as opposed to range checking enabled, consider the vectorization of "Make trapezoid buckets" (MTB) and "Empty trapezoid buckets" (ETB). With range checking disabled, MTB took 72.08 ms to execute, while ETB took 60.89 ms. With range checking enabled, MTB took 232.00 ms to execute, while ETB took 293.28 ms. With range checking enabled, MTB and ETB took almost four times longer to execute.

16. *Use unpacked arrays.* Minimize the use of operations on packed arrays, which the Pascal compiler cannot vectorize. Sometimes packed arrays must be used, however, and a solution to this problem is to operate on unpacked arrays and pack them later. (The Ultra frame buffer requires packed image arrays).

17. *Take advantage of large memory availability.* Most supercomputers have an enormous memory capacity. Trade memory for execution time. Avoid dynamic memory allocation requiring pointers, since the stride is unpredictable, and use large array structures. Use buckets when appropriate to accumulate vectors on which to operate.

18. *Use matrix solutions when possible.* Computer graphics requires matrix algebra for many data manipulations (such as rotations, scalings, translations, and other transformations). Attempt to express solutions in matrix form, if one exists, before vectorizing. Solutions which can be expressed in matrix algebra form are some of the easier to implement in vector.

19. *Realize that results are usually data sensitive.* Just because a vectorized program produces a speedup for one set of data, it may not produce a speedup for another set, especially if one set of data is characterized by long vectors, and another by short vectors. Collecting statistics on data characteristics may shed light on why speedup is not occurring.

20. *Do not get frustrated.* Vectorization is difficult, and requires time to learn. It is a new programming skill which is perhaps the most difficult of all programming skills to master.

4.4 High Level Design for a New Vectorized Z-buffer Algorithm

From experience gained in vectorizing the z-buffer, a new algorithm is presented which should provide a speedup over the current vectorized z-buffer.

1. Read the object and polygon descriptions.

2. Transform polygon local coordinates to world coordinates.

3. Transform polygons to triangles. Place triangles to be flat shaded, Gouraud shaded, and Phong shaded into 3 separate data structures.
4. For all frames to be generated do #5 - #14
5. * Transform the triangles to the proper view by rotating, scaling, or translating.
6. * Transform all triangle world coordinates to eye coordinates.
7. Clip all triangles to the screen.
8. * Decompose all triangles into trapezoids. Each triangle is decomposed into two trapezoids.
9. * Perform flat or Gouraud or Phong shading on all trapezoids.
10. * Perform perspective division on all trapezoids.
11. For each trapezoid do:
    (a) * Place trapezoids into height buckets.
    (b) If the current bucket contains 64 trapezoids, empty the bucket:
        i. * Determine the right and left edge values for position, (color, and normal if Gouraud or Phong shading is used) for all the trapezoids in the bucket. Now, each trapezoid in the bucket has been decomposed into scan-segments.
        ii. Place all scan-segments into a bucket for later interpolation.
12. Empty all height buckets which are not empty.
13. * Interpolate all scan-segments in vector.
14. Display the contents of the frame-buffer on the Ultra display.

Notice that steps seven through ten are performed on all the triangles at once, not on polygons taken one at a time as the design in Chapter 3 shows. This allows the designer to work with longer vectors than before.

Interpolation (no buckets)	Pascal		FORTRAN		Speedup	
	F-16	Dodeca	F-16	Dodeca	F-16	Dodeca
Edge	157.10	11.10	6.24	0.11	25.16	100.91
Scan-segment	507.80	1029.80	215.82	82.06	2.35	12.55
Total	664.90	1040.90	222.06	82.17	2.99	12.67

Table 4.7. Pascal and FORTRAN Interpolation Timings (ms)

4.5 Conclusion

Obviously, the Pascal code did not execute quickly enough for near real-time display. However, it does not follow that real-time performance is not possible with the existing Pascal compiler, but clearly, the code must be restructured if real-time performance is to be achieved; and yet restructuring may not produce the needed speedup. One option is to examine the Pseudo Cray Assembler Language (CAL) produced by the Pascal compiler for efficient instruction sequences. Unfortunately, the pseudo CAL output is not consistent with the actual machine instruction sequence produced by the compiler and therefore analysis of the pseudo CAL output cannot indicate reliably where the Pascal compiler might be generating inefficient object code (Conley, Sep 88).

The Pascal z-buffer indicates that the most time consuming portion of the code is edge and scan-segment interpolation, together consuming 64% of total display time for the vector version (615 ms) and 75% of the total display time for the scalar version (589 ms). Despite numerous attempts at restructuring these two routines, speedup has been elusive, so these routines were recoded in FORTRAN, given Cray FORTRAN is a more efficient compiler than the Pascal compiler. Estimates of the relative performance of FORTRAN to Pascal can be obtained by recoding edge and scan-segment interpolation routines in FORTRAN and comparing the results to those produced by the Pascal compiler. While it is difficult to determine whether the speedup is due to vectorization or due to using the more efficient FORTRAN compiler instead of the Pascal compiler, the results are astounding. Translating the Pascal edge and scan-segment interpolation routines into FORTRAN resulted in the timings, performed for both the F-16 and the "DodecaIcosahederon," shown in Table 4.7.

The FORTRAN version performed the edge and scan-segment interpolation routines in 222.06 ms for the F-16. Experimenting with the FORTRAN edge and scan-segment

interpolation routines by modifying data structures and restructuring the control flow may produce further speedup. With such superb performance obtained merely by translating Pascal into FORTRAN, the other sections of code should be written in FORTRAN, instead of Pascal, according to the design language presented in this chapter. Nonetheless, both of these routines must execute in less than 100 ms for near real-time performance. While 100 ms may be difficult to achieve solely by vectorization using FORTRAN, rewriting the critical z-buffer routines in Cray assembler may produce the desired speedup.

V. Conclusions

5.1 Introduction

This chapter covers the recommendations for follow-on thesis research, the economics of real-time image display, and concludes by addressing the thesis statement formulated in Chapter 1.

5.2 Recommendation for Future Thesis Research

This thesis can be enhanced in at least two ways:

1. Restructure the scalar algorithm to permit further vectorization, according to the z-buffer high-level design presented in Chapter 4. Use the FORTRAN programming language.
2. Parallelize the z-buffer software by using all four of the Cray-2's processors.

A combination of vectorization and parallelization is likely to produce speedups greater than either can achieve alone. Restructuring the scalar algorithm was already sufficiently discussed in Chapter 4. A parallel z-buffer algorithm is mentioned in at least four articles: (Fiume and Fournier, 1983), (Parke, 1980), (Kaplan and Greenberg, 1979), and (Weinberg, 1981). Two methods for implementing the code in parallel are:

1. Processor Pool.
2. Screen-space Division.

Each is discussed briefly in the following sections, with the intent of providing ideas for future research.

5.2.0.1 Processor Pool One of the four processors would act as an executive, parcelling polygons to one of three processors in the available processor pool. Polygon decomposition into trapezoids, and edge and scan-segment interpolation would proceed normally. When these calculations cease, the processor is placed back into the available

processor pool, awaiting assignment by the executive. The maximum speedup for this method is three. A major disadvantage to this method results from shared memory conflicts. It is possible that two or more processors may try to update simultaneously either the depth-buffer or the refresh-buffer, and preventing all but one of the processors from simultaneous access slows down execution. One advantage is approximately even processor load.

5.2.0.2 Screen Space Division In this method, the screen is divided into fourths and each of the four processors is assigned responsibility for displaying the proper part of the image on its portion of the screen. Each processor is given a copy of the entire object description, and it clips the object polygons to its screen portion. This method is not susceptible to several processors attempting to access identical memory cells in the z-buffer or refresh buffer, since each processor is responsible for its own z-buffer and refresh buffer. However, it is vulnerable to uneven processor load. If the object occupies only a certain part of the screen, one or more of the processors may be idle for significant amounts of time (or altogether idle). This can be minimized by carefully subdividing the screen (interleaving processor responsibilities over the screen), but the more complex the screen shape for which each processor is responsible, the more time that is spent clipping the object polygons to the screen shape. This method also complicates antialiasing, due to the communication of pixel color values from one processor to another for those pixels which border on two or more processors' screen area. Another consideration is that this method requires four copies of the object description, which can occupy a significant amount of memory. Maximum speedup due to this type of parallelization is four.

5.2.0.3 Options for Real-time Display Regardless of the method chosen, parallelization may negatively affect vectorization. One disadvantage of the screen space division method is that vector length decreases when scan-segments are broken into pieces for interpolation by different processors. Additionally, if bucketized edge or scan-segment interpolation is used, the number of height and length buckets quadruple for the four processors and each bucket may contain fewer scan-segments to interpolate in vector. Furthermore, the maximum speedup is four, so even if parallelization is exploited, real-time

performance may not be achieved. For a ten frame per second image display, the vectorized program must be capable of displaying the image in no more than 400 ms, and a speedup of four must be obtained by parallelization. An added disadvantage of parallelization is expense. While vectorization attempts to use each processor more efficiently, parallelization requires the use of more processor resources, which may be perceived unfavorable by other Cray-2 users competing for these resources.

5.3 The Economics of Real-time Image Display

Examining the economics of generating real-time images using the Cray-2 is not a primary objective of this thesis, but one cannot help but wonder whether the attempt is worthwhile. Certainly this thesis has provided some experience and insight into vectorizing algorithms, which is important and useful information if further research of real-time image generation using the Cray-2 is continued. What is not so obvious is whether the Cray-2 should be used for real-time image generation. Clearly, using the Cray-2 for this purpose is expensive, especially if all four processors are used. While the Cray-2 with the Ultra frame buffer is an enviable tool for graphics algorithm development and research, for those scientists not in the business of creating graphics algorithms, but just using them, the Cray-2 with the Ultra frame buffer is an expensive tool for graphics, especially with the advent of the computer graphics workstation. This does not lessen the importance of developing efficient algorithms targeted for the supercomputer, for there may be no option but to use the Cray-2, and speedy, efficient graphics display is highly desirable. There are cases when the supercomputer is absolutely essential for collecting and reducing scientific data. Downloading this data to a computer graphics workstation may be prohibitively expensive and there may be no other option than to use the Cray-2 for generating images.

However, excluding the economics of using the supercomputer from a decision to display data is unwise. Dedicated use of a multimillion dollar supercomputer is wasteful if a $100,000 workstation can do the job in the required time. Reserving the Cray-2 for graphics work when it is not feasible to download data from the supercomputer to a graphics workstation, or where flexibility in changing display algorithm is desired, should be examined. Otherwise, consideration should be given to using another, less expensive

computer.

It is also interesting that as this thesis was well underway, Titan, Ardent, and Silicon Graphics introduced machines that display images in real-time using the z-buffer algorithm; at a relatively low cost – around $100,000. All three companies used vector pipeline technology for initial data transformations, perspective division, and lighting, but in all three cases, specific VLSI hardware was developed for both edge and scan-segment interpolation. Could these companies have discovered already that general purpose hardware (vector pipelines) are advantageous only for coordinate transformation, perspective division, and lighting calculations, while specific VLSI graphics engines were required for edge and scan-segment interpolation? One can only speculate. However, using these machines is not without its disadvantages. Downloading display data from the supercomputer over low bandwidths to the workstations, data format restrictions, and lack of flexibility to change the display software are significant disadvantages which the Cray-2 does not have.

If real-time display is to be achieved, a combination of speedup techniques must be used. The following are options which may provide this speedup if correctly implemented:

1. Effective vectorization of the z-buffer code.
2. Coding the algorithm in FORTRAN or perhaps assembler.
3. Parallelization of the z-buffer software.
4. Replacing the z-buffer algorithm with another, more appropriate hidden-surface algorithm that is better suited to the Cray-2 architecture.

5.4 Conclusion

The purpose of this thesis was to gain experience and insight into vectorizing software. The heuristics presented in Chapter 4 can be applied to the vectorization of other computer software. One important observation is that *explicit vectorization* can obscure the clear function of software and decreases software readability. Thus, transforming scalar software into vector software using explicit vector syntax should be done sparingly, only when absolutely essential to meet performance requirements. However, efficient vectorizing compilers should be used to automatically vectorize software.

This thesis has shown that vectorization is a powerful means of increasing software efficiency. While real-time performance was not achieved, it does not follow that it is not possible, but it has been demonstrated that it is difficult to achieve. Consideration must be given to: 1) compiler efficiency, 2) supercomputer architecture, 3) type of data to be displayed, and 4) display algorithm. Clearly, when attempting to design efficient software if any one of these four areas is not optimized, the entire display performance will suffer. Originally, the code was developed with the assumption that the Pascal compiler could be relied upon to generate efficient code, but this was a faulty assumption. What has become increasingly obvious is that high level languages provide facilities to investigate differences in algorithm design, to weed out poor designs, and to provide rough estimates of performance, but caution must be exercised when the high level languages are used to make conclusions at low levels of detail, such as the efficiency of edge or scan-segment interpolation. At these levels, the software designer is susceptible to inefficient compiler design. Indeed, real-time performance is a difficult goal which will require further examination and time consuming analysis of the compiler used and the object code generated. It is not sufficient to merely rely upon good algorithm design to provide the speedups. When pushing the limits of computer hardware and software technology, it is essential that all four areas be given ample consideration.

Appendix A. Pascal Vector Syntax Overview

Pascal vector syntax makes extensive use of the slice expression. A slice selects certain array indices for processing. For example:

```
for j := 1 to 64 do
   a[i] := 0;
```

is equivalent to:

```
a[1..64] := 0;
```

The indices "1..64" could have been variables whose values were determined at execution time, or they could have been constants. Vector syntax can select multidimensional arrays. For example:

```
for i := 1 to 64 do
   for j := 1 to 256 do
      a[i,j] := 0;
```

is equivalent to:

```
a[1..64,1..256] := 0;
```

Stride can be specified in discrete increments for multidimensional arrays. If no stride is specified, the default stride is one. A one dimensional array with a stride of eight is written:

```
a[1..64,8] := 0;
```

Elements a[1], a[9], a[17], a[25], a[33], a[41], a[49], and a[57] receive the value 0. The others remain uninitialized. One final concept needs to be addressed. If an array were declared:

```
a : array[1..64] of integer;
```

then the following would be equivalent statements:

```
a[1..64] := 0; or
a        := 0;
```

A more detailed discussion of Pascal vector syntax is presented in the Cray Computer Systems *Pascal Reference Manual*, Chapter 6, Array Processing.

Bibliography

1. Akeley, Kurt and Tom Jermoluk. "High Performance Polygon Rendering," *ACM Computer Graphics, Vol 22, 4*: 239-246 (Aug 1988).
2. Apgar, Brian, Bersack, Bret, and Mammen, Abraham. "A Display System for the Stellar Graphics Supercomputer Model GS1000," *ACM Computer Graphics, Vol 22, 4*: 255-262 (Aug 1988).
3. Ardent Computer. *The Titan Graphics Supercomputer - A System Overview.* Ardent Computer Corp. 1988.
4. Bell, C. Gordon, Mirander, Glen S., and Rubinstein, Jonathan J. "Supercomputing for One," *IEEE Spectrum, Vol 25, 4*: 46-50 (April 1988).
5. Bishop, Gary, and Weimer, David M. "Fast Phong Shading," *ACM Computer Graphics, Vol 20, 4*: 103- 106 (Aug 1986).
6. Bouchier, Charles M. "Reaping the Rewards of Vector Processing," *Computer Design, Vol 25, 16*: 79-84 (Sep 1986).
7. Buzbee, B. L. "A Strategy for Vectorization," *Parallel Computing, Vol 3 3*: 187-192 (Jul 86).
8. Conley, Robert, Chief of Graphics Research. Personal Correspondence. Air Force Weapons Laboratory (AFWL/SCPS), Kirtland Air Force Base, NM, 1988.
9. Cray Research, Inc. *Cray-2 Computer System Functional Description HR-2000.* Cray Research, Inc., Technical Operations Building, Chippewa Falls, WI 54729, July 1987.
10. Cray Research, Inc. *Cray Computer Systems Pascal Reference Manual SR-0060.* Cray Research, Inc., Technical Operations Building, Chippewa Falls, WI 54729, January 1986.
11. Diede, Tom, Hagenmaier, Carl F., Miranker, Glen S., Rubinstein, Jonathan J., and Worley, William S. Jr. "The Titan Graphics Supercomputer Architecture," *IEEE Computer, Vol 21, 9*: 13-30 (September 1988).
12. Dyer, Scott and Scott Whitman. "A Vectorized Scan-Line Z-Buffer Rendering Algorithm," *IEEE Computer Graphics and Applications*,:34-45 (Jul 1987).
13. Fisher, David A. "DoD's Common Programming Language Effort," *Computer, 3*: 24-33 (March 1978).
14. Fiume, Eugene and Alain Fournier. "A Parallel Scan Conversion Algorithm with Anti-aliasing for a General Purpose Ultracomputer," *ACM Computer Graphics, Vol 17, 3*: 141-149 (July 1983).
15. Foley, J. D., and Van Dam, A. *Fundamentals of Interactive Computer Graphics.* Reading, MA: Addison-Wesley Publishing Co., 1982.
16. Frenkel, Karen A. "The Art and Science of Visualizing Data," *Communications of the ACM, Vol 31, 2*: 111- 121 (Feb 1988).
17. Goldapp, Michael. "Fast Scan-line Conversion using Vectorisation," *Parallel Computing, Vol 3, 2*: 141-152 (May 1986).

18. Greenberg, Donald P. "Coons Award Lecture," *Communications of the ACM, Vol 31, 2*: 123-151 (Feb 1988).

19. Higbie, Lee. "A Vector Processing Tutorial," *Datamation, Vol 29, 8*:180-200 (Aug 1983).

20. Kamrath, Anke. "How Cray-2 Users can Avoid Delays from Memory Conflicts," *Vector View, Vol 1, 3* (May 1988).

21. Kaplan, Michael and Greenberg, Donald P. "Parallel Processing Techniques for Hidden Surface Removal," *ACM Computer Graphics, Vol 13, 2*: 300-307 (Aug 1979).

22. McCormick, Bruce H., DeFanti, T.A., and Brown, M.D., Eds. "Visualization in Scientific Computing," *ACM Computer Graphics, Vol 21, 6* (Nov 1987).

23. Parke, Frederic I. "Simulation and Expected Performance Analysis of Multiple Processor Z-buffer Systems," *ACM Computer Graphics, Vol 14, 3*: 48-56 (July 1980).

24. "Special Report: Visualization in Scientific Computing - A Synopsis," *IEEE Computer Graphics and Applications*, 61-70 (Jul 1987).

25. Staudhammer, John. "Directions in Computer Graphics Architecture," *Proceedings of the Work shop on Future Directions in Computer Architecture and Software*. 296-298. Charleston, SC: US Army Research Office: May 5-7, 1986.

26. Staudhammer, John. "Supercomputers and Graphics," *IEEE Computer Graphics and Applications,*: 24-25 (Jul 1987).

27. Stone, Harold S. *High-Performance Computer Architecture*. Reading, MA: Addison-Wesley Publishing Co., 1987.

28. Sutherland, Ivan E., Sproull, Robert F., and Schumacker, Robert A. "A Characterization of Ten Hidden-Surface Algorithms," *Computing Surveys, Vol 6, 1*: 1-55 (March 1974).

29. Swanson, Roger W., Thayer, Larry J. "A Fast Shaded-Polygon Renderer," *ACM Computer Graphics, Vol 20, 4* (Aug 1986).

30. Weinberg, Richard. "Parallel Processing Image Sysnthesis and Anti-Aliasing," *ACM Computer Graphics, Vol 15, 3*: 55-62 (Aug 1981).

31. Yourdon, Edward. *Managing the Structured Techniques*. Englewood-Cliffs, NJ: Prentice-Hall Inc., 1979.

Vita

Roy Donehower was born [redacted] He graduated from Radford High School in Honolulu, Hawaii in 1980. After High School, he entered the United States Air Force Academy and graduated in May of 1984 as a distinguished graduate with honors, earning a Bachelor of Science degree and commissioned as a Second Lieutenant in the USAF. Prior to coming to the Air Force Institute of Technology (AFIT), he served as a computer scientist for the Embedded Computer Resources Division, Headquarters Military Airlift Command, Scott AFB, Illinois. He attended AFIT from June 1987 to December 1988, and graduated with a Master of Science in Computer Engineering degree.

[redacted]

Recent developments in scientific computing have prompted the need for supercomputer graphics research. These developments include the requirement to visualize large amounts of data which are processed by the supercomputer, preferably by real-time image generation. Unfortunately, most currently used graphics algorithms are not optimized for vector computers. This thesis involves the design and implementation of a hidden-surface removal algorithm for the Cray-2 vector supercomputer, with the goal of real-time image display.

A z-buffer hidden-surface algorithm, written in Pascal, was vectorized and implemented on the Cray-2. Special attention was directed toward the methodology of algorithm and data structure design to exploit the Cray-2 architecture. Timing studies comparing the vector version to the equivalent scalar version showed that while the Pascal vector code produced localized speedups, the vector code was less efficient than the scalar code. When critical portions of the code were translated from Pascal to FORTRAN, significant speedup was achieved, indicating that the FORTRAN compiler generates more efficient object code than the Pascal compiler. However, real-time performance was not achieved. Based on the knowledge gained from vectorizing the z-buffer algorithm, vectorization heuristics were discovered and a new algorithm which should provide further speedup is presented.

Experience from the thesis research indicates that when attempting to achieve real-time performance, consideration must be given to compiler efficiency, supercomputer architecture, type of data displayed, and display algorithm. Relying only upon good algorithm design may not produce adequate performance, and if any of these four areas is not properly addressed, performance suffers. Another important observation is that vectorization is difficult and can obscure the clear function of software, decreasing software readability and perhaps maintainability. Thus, transforming scalar software into software structured for vector operations should be done sparingly: only when absolutely essential to meet performance requirements. However, despite its difficulties, vectorization is a powerful way to increase software efficiency.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): AFIT/GCS/ENG/88D-3	5. MONITORING ORGANIZATION REPORT NUMBER(S)
6a. NAME OF PERFORMING ORGANIZATION: School of Engineering; 6b. OFFICE SYMBOL (If applicable): AFIT/ENG	7a. NAME OF MONITORING ORGANIZATION
6c. ADDRESS (City, State, and ZIP Code): Air Force Institute of Technology, Wright-Patterson AFB OH 45433-6583	7b. ADDRESS (City, State, and ZIP Code)
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: AFWL; 8b. OFFICE SYMBOL (If applicable): SCP	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS (City, State, and ZIP Code): Kirtland AFB, NM 87117	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.
11. TITLE (Include Security Classification): See box 19	
12. PERSONAL AUTHOR(S): Roy Donehower, B.S., Capt, USAF	
13a. TYPE OF REPORT: MS thesis; 13b. TIME COVERED: FROM _____ TO _____; 14. DATE OF REPORT (Year, Month, Day): 1988 December; 15. PAGE COUNT: 97	
16. SUPPLEMENTARY NOTATION	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	
12	06		Supercomputer, Computer Graphics, THESIS. (JES)
12	05		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

Title: A VECTORIZED HIDDEN-SURFACE ALGORITHM IMPLEMENTED ON THE CRAY-2 SUPERCOMPUTER

Thesis Advisor: Phil Amburn, Major, USAF
Professor of Computer Systems

Approved for public release IAW AFR 190-1
12 Jan 1989

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT: ☐ UNCLASSIFIED/UNLIMITED ☑ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION: UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL: Phil Amburn, Major, USAF	22b. TELEPHONE (Include Area Code): (513) 255-3576; 22c. OFFICE SYMBOL: AFIT/ENG

DD Form 1473, JUN 86 — Previous editions are obsolete. — SECURITY CLASSIFICATION OF THIS PAGE: UNCLASSIFIED